

# PARADIGM PROJECT

Logic Programming Associates Ltd.

**micro-PROLOG 3.0  
Programmer's Reference Manual**

CP/M Version \*

F.G. McCabe  
K.L. Clark

Third Edition

This manual describes the micro-PROLOG system from the programmer's point of view. It describes the syntax of micro-PROLOG, the various built-in features, and how to interact with the system. Also included is a chapter which describes the machine code level interface, and how to augment the system with built-in predicates. In so far as it is specific to any particular micro-PROLOG implementation, this manual describes the CP/M version.  
The specification of micro-PROLOG is subject to change without notice.

September, 1980  
May, 1981  
February 1983

(c) F.G. McCabe 1980  
(c) Logic Programming Associates Ltd., 1981, 1983

\* CP/M is a registered trade mark of Digital Research Inc.  
micro-PROLOG is a trade mark of LPA Ltd.

## Acknowledgements

The authors would like to express their gratitude to R.A.Kowalski without whom this project would have been impossible, and to C.D.S.Moss for many fruitful discussions on the syntax and machine level interface of micro-PROLOG.

## Contents

Chapter		Page
1.	Introduction . . . . .	1-1
1.1	micro-PROLOG . . . . .	1-1
1.2	Extensibility . . . . .	1-2
1.3	Version 3 . . . . .	1-2
1.4	User preparation . . . . .	1-3
1.5	Notation conventions . . . . .	1-3
2.	Syntax of micro-PROLOG . . . . .	2-1
2.1	Character set . . . . .	2-1
2.2	Numbers . . . . .	2-1
2.3	Constants . . . . .	2-2
2.4	Variables . . . . .	2-4
2.4.1	A note on separators . . . . .	2-4
2.5	Lists . . . . .	2-4
2.5.1	The list constructor ! . . . .	2-5
2.5.2	List patterns . . . . .	2-6
2.6	Atoms . . . . .	2-7
2.7	Clauses . . . . .	2-8
2.8	Comments . . . . .	2-9
2.8.1	Comment conditions . . . . .	2-9
2.8.2	Comment clauses . . . . .	2-9
2.9	Meta-variable . . . . .	2-10
2.9.1	Meta-variable as Predicate Symbol . . . . .	2-10
2.9.2	Meta-variable as an atom . . . . .	2-10
2.9.3	Meta-variable as the rest of the body of a clause . . . . .	2-11
2.10	Lexical syntax . . . . .	2-11
2.10.1	Token boundaries . . . . .	2-11
2.10.2	Special tokens . . . . .	2-12
2.10.3	Alpha-numeric tokens . . . . .	2-12
2.10.4	Number tokens . . . . .	2-12
2.10.5	Graphic tokens . . . . .	2-12
2.10.6	Quoted strings . . . . .	2-13
2.10.7	The lexical rules . . . . .	2-13
3.	Interacting with the micro-PROLOG system . . . . .	3-1
3.1	The micro-PROLOG distribution disk . . . . .	3-1
3.1.1	Invoking micro-PROLOG . . . . .	3-2
3.2	Keyboard control and line editor . . . . .	3-3
3.2.1	Edit mode . . . . .	3-4
3.2.2	Type ahead . . . . .	3-6
3.2.3	Multi-line terms and right bracket prompts . . . . .	3-6
3.3	Supervisor commands . . . . .	3-7
3.3.1	Entering clauses . . . . .	3-7
3.3.2	The LIST command . . . . .	3-8
3.3.3	The query command ? . . . . .	3-8
3.3.3.1	Interrupts . . . . .	3-9
3.3.3.2	Other uses of the query command . . . . .	3-9
3.3.3.4	Unary relations as commands . . . . .	3-9

3.3.5 Multi-argument commands . . . . .	3-10
3.3.6 LOAD and SAVE commands . . . . .	3-11
3.3.6.1 File names . . . . .	3-11
3.3.7 The DICT relation . . . . .	3-12
3.3.8 The KILL command . . . . .	3-13
3.3.9 Exiting the system . . . . .	3-13
3.4 Pragmatic Considerations for Programmers . . . . .	3-14
4. Utility modules . . . . .	4-1
4.1 Tracing Execution . . . . .	4-1
4.2 Spypoint tracing . . . . .	4-3
4.2.1 The spy command . . . . .	4-3
4.2.2 The spying relation . . . . .	4-4
4.2.3 The spying command . . . . .	4-5
4.2.4 The unspy command . . . . .	4-5
4.3 The micro-PROLOG structure editor . . . . .	4-5
4.3.1 Edit Commands . . . . .	4-6
4.3.2 Cursor Movement Commands . . . . .	4-8
4.3.3 Edit change Commands . . . . .	4-10
4.3.4 Restructuring Lists . . . . .	4-12
4.3.5 Further Extension . . . . .	4-12
4.4 Editing modules . . . . .	4-13
4.4.1 The unwrap command . . . . .	4-13
4.4.2 The wrap command . . . . .	4-14
4.5 Using external relations - the EXREL utility . . . . .	4-15
4.5.1 external . . . . .	4-15
4.5.2 The RPRED relation . . . . .	4-16
4.5.3 The open command and the opened relation . . . . .	4-16
4.5.4 The close command . . . . .	4-17
4.5.5 The internal command . . . . .	4-17
4.5.6 The LISTEX, LISTFILE commands . . . . .	4-17
4.5.7 The listex, listfile commands . . . . .	4-17
4.5.8 Changing the definitions of external relations . . . . .	4-17
5. Simple PROLOG . . . . .	5-1
5.1 Syntax of sentences accepted by SIMPLE . . . . .	5-2
5.2 The relations and commands defined by SIMPLE . . . . .	5-4
5.2.1 Add . . . . .	5-4
5.2.2 List . . . . .	5-4
5.2.3 Delete . . . . .	5-5
5.2.4 Kill . . . . .	5-5
5.2.5 Does . . . . .	5-6
5.2.6 Which . . . . .	5-6
5.2.7 One . . . . .	5-7
5.2.8 Save . . . . .	5-7
5.2.9 Load . . . . .	5-7
5.2.10 Accept . . . . .	5-8
5.2.11 Edit . . . . .	5-8
5.2.12 APPEND, ON, true-of . . . . .	5-9
5.2.13 CONS, @, #, =, function . . . . .	5-10
5.2.14 *, %, +, -, PROD . . . . .	5-10
5.3 Using expressions in sentences . . . . .	5-10
5.3.1 Expression conditions . . . . .	5-10
5.3.2 Equality conditions . . . . .	5-11
5.3.3 Syntax of expressions . . . . .	5-12
5.3.4 Warning on the use of ! in expressions . . . . .	5-15
5.3.5 When the expression handler is not needed . . . . .	5-16
5.4 The error handler errmsgs-mod . . . . .	5-17

5.5 Using the relation is-told . . . . .	5-18
5.5.1 Example use of is-told . . . . .	5-19
5.5.2 Killing told-mod . . . . .	5-20
5.6 Using external relations the EXREL utility . . . . .	5-20
5.7 Tracing SIMPLE queries using SIMTRACE . . . . .	5-20
5.7.1 The relations exported by simtrace-mod . . . . .	5-21
6. The MICRO extension to the supervisor . . . . .	6-1
6.1 The relations exported by micro-mod . . . . .	6-1
6.1.1 add . . . . .	6-2
6.1.2 delete . . . . .	6-3
6.1.3 edit . . . . .	6-3
6.1.4 cedit . . . . .	6-3
6.1.5 kill . . . . .	6-4
6.1.6 list . . . . .	6-4
6.1.7 load . . . . .	6-4
6.1.8 save . . . . .	6-4
6.1.9 reserved . . . . .	6-4
6.1.10 space . . . . .	6-4
6.1.11 ask . . . . .	6-5
6.1.12 which, all . . . . .	6-5
6.1.13 one . . . . .	6-6
6.1.14 accept . . . . .	6-6
6.1.15 APPEND, ON, true-of . . . . .	6-7
6.1.16 CONS, @, #, =, function . . . . .	6-7
6.1.17 *, %, +, -, PROD . . . . .	6-7
6.1.18 dir . . . . .	6-8
6.2 Expressions in MICRO clauses . . . . .	6-8
6.2.1 The f relation . . . . .	6-9
6.2.2 The = relation . . . . .	6-10
6.2.3 Syntax of expressions . . . . .	6-10
6.2.4 Killing expran-mod . . . . .	6-11
6.3 The error handler errtrap-mod . . . . .	6-12
6.3.1 Example error recovery . . . . .	6-13
6.4 Using the is-told relation . . . . .	6-13
6.4.1 Example use of is-told . . . . .	6-14
6.4.2 Killing told-mod . . . . .	6-14
6.5 External relations . . . . .	6-14
6.6 Tracing and Structure editing . . . . .	6-14
7. Built-in Programs . . . . .	7-1
7.1 Arithmetic Relations . . . . .	7-2
7.1.1 SUM . . . . .	7-2
7.1.2 TIMES . . . . .	7-2
7.1.3 LESS . . . . .	7-3
7.1.4 INT . . . . .	7-3
7.1.5 SIGN . . . . .	7-4
7.2 String operations . . . . .	7-4
7.2.1 LESS . . . . .	7-4
7.2.2 STRINGOF . . . . .	7-5
7.2.3 CHAROF . . . . .	7-5
7.3 Console Input/Output . . . . .	7-6
7.3.1 R . . . . .	7-6
7.3.2 P . . . . .	7-7
7.3.3 PP . . . . .	7-7
7.3.4 RFILL . . . . .	7-8
7.4 File I/O . . . . .	7-9



7.4.1 OPEN	7-9
7.4.2 CREATE	7-9
7.4.3 CLOSE	7-9
7.4.4 READ	7-10
7.4.5 WRITE	7-10
7.4.6 W	7-10
7.4.7 INTOK	7-10
7.4.8 SEEK	7-11
7.4.9 Special file names	7-12
7.5 File operations	7-13
7.5.1 LOGIN	7-13
7.5.2 ERA	7-14
7.5.3 REN	7-14
7.5.4 DIR	7-14
7.6 Record input/output	7-15
7.6.1 FWRITE	7-16
7.6.2 FREAD	7-17
7.6.3 Relations defined by files of records	7-19
7.6.4 Mixing records and terms	7-20
7.7 Type Predicates	7-20
7.7.1 NUM	7-20
7.7.2 INT	7-20
7.7.3 CON	7-20
7.7.4 LST	7-21
7.7.5 SYS	7-21
7.7.6 VAR	7-21
7.8 Logical operators	7-21
7.8.1 OR	7-21
7.8.2 NOT	7-22
7.8.3 IF	7-24
7.8.4 EQ	7-25
7.8.5 ?	7-25
7.8.6 FORALL	7-26
7.8.7 ISALL	7-26
7.8.8 I	7-28
7.9 Data Base operations	7-29
7.9.1 CL	7-29
7.9.2 ADDCL	7-31
7.9.3 DELCL	7-32
7.9.4 Undoing the effect of ADDCL or DELCL	7-32
7.9.5 KILL	7-33
7.10 Library procedures	7-33
7.10.1 LIST	7-33
7.10.2 SAVE	7-34
7.10.3 LOAD	7-35
7.11 Module Construction Facilities	7-35
7.11.1 CMOD	7-37
7.11.2 CRMOD	7-37
7.11.3 OPMOD	7-38
7.11.4 CLMOD	7-38
7.12 Miscellaneous Predicates	7-38
7.12.1 DICT	7-39
7.12.2 QT	7-39
7.12.3 /	7-39
7.12.4 FAIL	7-40
7.12.5 ABORT	7-40
7.12.6 /#	7-40
7.12.7 SPACE	7-41
7.12.8 <SUP>	7-41

8. Adding assembler coded subroutines	8-1
8.1 Data registers	8-2
8.2 Value cells and terms in micro-PROLOG	8-3
8.2.1 micro-PROLOG internal data structures	8-4
8.2.2 Warning	8-5
8.3 Type tree	8-7
8.4 Predicate symbol declaration	8-8
8.5 Inserting a program	8-8
8.6 Extending micro-PROLOG	8-9

Appendices	
A. Error messages and error handling	A-1
B. Useful addresses	B-1
C. Changing the lexical rules	C-1
D. System requirements	D-1
References	R-1

## 1. Introduction

This manual describes the micro-PROLOG logic programming system. PROLOG is a computer language based on predicate logic, in particular the clausal form of logic and resolution inference [Robinson 1965, 1979]. micro-PROLOG is an interpreted version of PROLOG tailored for interactive use on micro-computers. It is the end product of three years of continuous development in the field of PROLOG systems on micros. It is a mature system, which although it runs on a micro computer, does not sacrifice any significant features of the PROLOG systems on large main-frame computers. Indeed, it contains many more features than some of the main-frame systems.

The first PROLOG (which stands for PROgramming in LOGic) was implemented in 1972 in Marseilles by Colmerauer and Rousset [Colmerauer 1973] as an interpreter in the medium level programming language ALGOL-W. A more efficient and improved interpreter was then built in 1973 [Rousset 1975], this time in FORTRAN. This implementation reached a wide audience in countries as far afield as Poland, Hungary, U.S.A., Canada, Sweden, Portugal, Belgium and the U.K. Building on and extending the implementation techniques of the Marseilles PROLOG, there have been several other implementations, the principal ones being the Edinburgh DEC-10 PROLOG [Warren et al. 1978], Waterloo PROLOG [Roberts 1977], the Hungarian M-PROLOG [Szerepi 1982] and a new implementation from Marseilles [Kanoui & Van Caneghem 1980]. The Edinburgh implementation incorporates a compiler, the first to be written for the language. Recently, interest in PROLOG has been stimulated by the Japanese decision to use it as the core language for which they will design their fifth generation computers. The Japanese interest derived from the 'academic export' of the DEC-10 implementation to Japan.

### 1.1 micro-PROLOG

To allow a compact and efficient implementation, mostly written in assembler, the basic syntax and the built-in supervisor of micro-PROLOG are quite rudimentary. The basic syntax is very close to the list syntax of LISP and the supervisor is a small micro-PROLOG program that provides only the necessary program development and query facilities. The most important feature of the supervisor is that it is extensible. By loading other micro-PROLOG programs, wrapped up as modules, one can considerably enhance the program development and query facilities of the built-in supervisor. Several such supervisor extension modules are provided with the micro-PROLOG system.

Modules are an important program structuring facility of micro-PROLOG which is one of a very few implementations of PROLOG to support modules. micro-PROLOG modules are named collections of relation definitions that communicate with other programs via import/export name lists. All other names used in the module are local to the module. This localised name feature enables modules developed at different times, or by different programmers, to be used together without fear of a name clash. Finally, exported names of relations defined in the module can be used in other

## 1. Introduction

programs as though they were primitive relations of micro-PROLOG. Hence the role of modules in extending the facilities of the supervisor.

One collection of supervisor extension modules, in the file SIMPLE.LOG, accepts program clauses and queries expressed in the sugared, easy to read sentence syntax described in the Primer. It compiles these into the basic syntax. It also incorporates an interactive program text editor which first decompiles program clauses so that they can be edited in the sentence syntax. The facilities of SIMPLE and the syntax of SIMPLE programs are fully described in Chapter 5.

Another suite of program modules, in the file MICRO.LOG, provides all of the program development and query modes of SIMPLE but for programs and queries written in the basic syntax augmented with functional expressions. The facilities provided by the MICRO modules are described in Chapter 6.

Other utility modules are provided in the files:

```
EDITOR.LOG {a structure editor},  
TRACE.LOG {a full trace package},  
SPYTRACE.LOG {allows spy-tracing of individual programs},  
MODULES.LOG {facilitates construction and editing modules},  
EXREL.LOG {a package that allows disc files to be used as  
virtual memory for programs},  
SIMTRACE.LOG {a trace package for use with SIMPLE},  
DEFTRAP.LOG {a simple error handler},  
ERRORTRAP.LOG {an error handling and recovery utility}.
```

The EDITOR, TRACE, SPYTRACE, MODULES and EXREL programs are described in Chapter 4. SIMTRACE and DEFTRAP are described in Chapter 5 along with SIMPLE, and ERRORTRAP is described in Chapter 6 along with MICRO.

All of these utility modules can be optionally loaded as extensions to the built-in supervisor and deleted when they are no longer needed. Beginning micro-PROLOG programmers usually start by using the SIMPLE extension, in the manner described in the Primer, and then when they are completely conversant with all its facilities, they move over to using MICRO or their own micro-PROLOG implemented aid to program development. Experienced micro-PROLOG programmers, like LISP programmers, have no problem using the micro-PROLOG basic syntax.

### 1.2 Extensibility

A design objective of micro-PROLOG was to build a kernel system that could easily be extended by the user and tailored to his requirements. The support of program modules allows for extension above, that is it allows extensions to the syntax and the supervisor commands to be implemented in micro-PROLOG. The system can also be extended below. New primitive relations can be added using a system provided interface to machine code programs. This interface is described in Chapter 8. The system provided primitive relations are fully described in Chapter 7.

### 1.3 Version 3

This reference manual is for version 3.0 of micro-PROLOG for CP/M 2.x Z80 computers. This is the fourth released version of micro-PROLOG. (The previous ones were 2.02, 2.11, and 2.12). In version 3.0 a large number of changes and improvements have

## 1. Introduction

been made to the system. Extra facilities include: floating point arithmetic, error trapping and user defined error recovery, record oriented input/output and the treatment of physical devices such as the console as special serial files. These special files can be written to and read from using the general purpose file I/O primitives.

Apart from these substantial enhancements to the micro-PROLOG system itself, the supporting micro-PROLOG software has been extensively rewritten and improved. A completely new supervisor extension (MICRO) has been introduced for those programmers who prefer the basic syntax to the syntax offered by SIMPLE. The SIMPLE for version 3.0 is a compatible enhancement of the SIMPLE described in the Primer. The extra features are fully described in Chapter 5. The major change is the ability to use arithmetic and functional expressions in definitions and queries. Such expressions can also be used in the definitions and queries accepted by MICRO.

One unfortunate side-effect of these changes is that micro-PROLOG 3 is somewhat larger than earlier versions, but it is a much more powerful system.

### 1.4 User preparation

To be able to exploit the full power of micro-PROLOG 3 we recommend that you study the Primer, doing all the exercises using the system. (Sample solutions are given in an appendix of the Primer.) Then read the whole of the manual. This effort to become fully conversant with all the facilities of micro-PROLOG will be well rewarded.

### 1.5 Notation conventions

The manual includes many examples of small micro-PROLOG programs to illustrate the use of some facility. All example programs are given in heavy type. Heavy type is also used when giving the name of a relation or module in the text.

In specifying the syntax of some micro-PROLOG construct the notation

<type>

is used to indicate that a field or sub-expression can be any instance of the indicated type. For example,

LOAD <file name>

is used to indicate that any allowed file name can be used in the LOAD command. <type1> and <type2> will be used when two uses of the same type can be different. The use of the different subscripts does not imply that they must be different.

One syntactic category, that of <term>, is used so often that the abbreviations t, t1, t2 etc. are used for <term>, <term1>, <term2>. The form

t1 ... tk

is used to indicate a sequence of k terms. Unless the condition k>0 is explicitly given, this form includes the case when there are no terms in the sequence. Again, the use of the different subscripts does not imply that the terms must be different, it

## 1. Introduction

implies only that they may be different.

To indicate the hitting of the RETURN or ENTER key <return> is used.

Finally, some example programs will be given with associated comments between "(," ")" brackets. They are not part of the program and should not be typed if you enter the program. micro-PROLOG does not accept comments in this form.

### WARNING

This manual is not an introduction to micro-PROLOG programming. It is intended to be used as a reference manual by the programmer who already has some knowledge of PROLOG programming to the extent covered by the companion micro-PROLOG Primer or the Clocksin & Mellish text [1981]. For an introductory text on the general ideas of logic programming we recommend the book "Logic for Problem Solving" [Kowalski 1979].

**The reader should also have some knowledge of the CP/M operating system, or be able to access a CP/M manual.**

### Trade marks

CP/M is a trade mark of Digital Research Inc., Z80 is a trade mark of Zilog Corp., micro-PROLOG is a trade mark of Logic Programming Associates Ltd.

## 2. Syntax of micro-PROLOG

Most mainframe PROLOG systems nowadays have very powerful grammars built into them; typically some kind of operator precedence grammar. While this might have been desirable, space limitations in a micro computer make this an expensive luxury if it is incorporated into the basic system.

Instead the basic syntax of micro-PROLOG is modelled on LISP syntax [McCarthy 1962]. If a richer syntax is desired it has to be supported by a front end micro-PROLOG program that compiles into the basic syntax. In this chapter we describe the basic syntax.

In Chapter 5 a more user friendly syntax that is compiled into the basic syntax by the SIMPLE extension for the supervisor is described. The SIMPLE supported syntax is just one option for a more elaborate syntax. Incidentally, the Primer calls the SIMPLE supported syntax the surface syntax and the basic syntax the internal syntax.

There are only four different kinds of syntactic objects that micro-PROLOG knows about: Numbers, Constants, Variables and Lists. They are all different kinds of Term.

The only data constructor in micro-PROLOG is the list constructor `l`. Lists constructed using `l` can be written in a specially condensed syntax as in DEC-10 PROLOG and LISP. Other PROLOG systems do allow other kinds of data constructors; these can be easily simulated (with no great loss of efficiency) using lists.

### 2.1 Character set

micro-PROLOG uses the 7 bit ASCII character set, together with extra characters, such as special graphics characters, that may be supported by the underlying machine. Characters are represented internally by 8 bit numbers in the range 1..126. The characters corresponding to 0 and 127 are not legal in micro-PROLOG, and are ignored if used.

### 2.2 Numbers

Numbers are integers or are floating point numbers.

A positive integer is written as a contiguous sequence of digit characters, with no leading sign character, eg. 0 30 1025 32767.

A negative integer is written with the leading sign character - contiguously followed by a positive integer. For example, -1 -30 & -32767 are all negative numbers. If a sign character does not have a positive integer contiguously following it, then it is not regarded as the sign of a number. Thus - on its own is a valid syntactic object, differing from any number. Integers must be in the range -32767 to 32767.

A floating point number can have up to 8 digits of precision, and an exponent in the range 10-127 to 10127. They are written in a fairly conventional notation; some example

floating point numbers are:

```
2.34      10.3e99      12.003E-100      -0.9
```

One and only one period must be present in a floating point number. As with an integer, it must start with a digit or a minus sign and a digit. The e exponent is optional, but if used it must be contiguous to the number and must be contiguously followed by an integer. The following are not floating point numbers:

```
.9      {starts with .}
3e-22   {no .}
34 e3   {space before e}
-.7e45  {no digit after -}
56e4.8  {exponent not an integer}
```

No matter how they are entered floating point numbers are always displayed in a standard format, which depends on the magnitude of the number. If the number is in the range  $-10^x < 10$  then it is displayed with the leading digit followed by the fractional part, as in:

```
4.2345678
```

Otherwise floating point numbers are displayed in the standard scientific notation:

```
9.99999999E99
```

but with only the significant digits displayed - trailing 0's after the . are suppressed.

As with integers, only negative floating point numbers have a sign character in front of them, positive floating point numbers do not have a sign character. If you enter +6.86 micro-PROLOG will interpret this as a sequence of two terms, the constant + and the positive number 6.86.

micro-PROLOG attempts to handle the conversion between integers and floating point numbers in a generally automatic and transparent way. In particular, integers and floating point numbers can be mixed in any way in arithmetic computations, and the results are represented as 16 bit integers if they can be. So, for example, the number entered as:

```
2.34e4
```

will be represented internally as the integer

```
23400
```

## 2.3 Constants

Constants are the simple unstructured objects of micro-PROLOG. They are used to name individuals such as fred, Al. They are also used to name relations such as member-of, father-of.

A constant is normally written as a alphabetical character (a letter), followed by a sequence of letters and digits (though see definition of variable below). This is similar to the way identifiers are written in conventional programming languages.

The - character also counts as an alphabetic character in a sequence of letters and digits. This can be used to split up long names with several English words. Constants can also be written using the non-alphanumeric characters such as ., !, etc. This kind of constant is written as any sequence of symbolic characters (or graphic characters supported by the machine) other than:

```
( ) [ ]
```

which have a special syntactic role, and

```
{ } [ ] < >
```

which are always interpreted as single character constants.

The symbols that can be used to form a symbolic constant include:

```
! & $ % ' - ^ _ \ @ ` ; : , . / + * ?
```

Thus,

```
** ! ?? -- :=
```

are all symbolic constants.

Finally, a constant can also be written as a quoted sequence of characters, in which case there are no restrictions on the characters that can be used in the constant. A quoted constant consists of a sequence of characters surrounded by the double quote character: ". Examples of quoted constants are:

```
"! "The man" "?ERROR?" "$100" "<SUP>"
```

A quoted constant can include a space, as in "The man", it can also include a new line or a tab. Outside a quoted string, these act as separators. Thus,

```
The man
```

is a sequence of two constants.

ASCII control characters can also be included in a quoted constant. This is useful for testing character input, or for positioning the cursor on a terminal which has cursor control. When printed using the micro-PROLOG P relation, a cursor control character in a quoted constant will move the cursor.

To put a control character in a quoted constant enter @ <letter> where <letter> is the letter of the key you would press in the control-key combination for the control character. Thus, to insert control-A enter @A. micro-PROLOG will convert this @A in the quoted constant into ASCII control code for control-A. When you list the program the occurrence of control-A in the constant will be displayed as @A.

In a quoted constant @ acts as an escape character causing micro-PROLOG to interpret the next character in a special way. To enter @ in a string you must use @@. Thus, "@A@" is the constant comprising two characters: control-A followed by @.

An @ character followed by the quote character means a quote character in the string.

```
"A quoted @ string"
```

has the text:

A quoted " string

Although constants can be of any length, only the first 60 characters of a constant are significant, and only these 60 characters are stored. This applies to quoted constants as well as other kinds of constant.

## 2.4 Variables

Variables are represented by alphanumeric names which consist of a single letter optionally followed by a contiguous positive integer which has the role of a subscript. The first character must be one of the variable prefix characters, which in standard micro-PROLOG are: x, y, z, X, Y & Z.

So, for example, x, X1 and Y30 are variable names since they consist of a variable prefix character contiguously followed (if at all) only by digits, whereas yes, x12c are not variable names. The integer which follows the variable prefix character should be in the range 1..127, otherwise two apparently distinct names will be mapped to the same variable.

(WARNING: x and x0 are the same variable because x is implicitly subscripted with 0. Similarly, x3 and x03 are the same variable because 3 and 03 are the same integer subscript. The subscript is not the digit sequence but the positive integer that the sequence denotes.)

When a term is read in, and a variable is recognised in the term, the variable is converted into a special internal form and the actual variable name used is discarded. All occurrences of the same variable name in the term are converted into the same internal form. However, when the term is printed the original name of the variables are not available. Instead, the variables in the term are displayed with names taken from the sequence X, Y, Z, x, y, z, X1, ..., z1, ... The first variable in the term is given the name X, the second is given the name Y, and so on. The set of variable prefix characters can be altered (see Appendix C).

### 2.4.1 A note on separators

Variables, constants and numbers are generally separated by one or more of the separator characters: space, carriage-return/new-line, tab. The actual number of separators is not important.

See Section 2.10 for a fuller description of micro-PROLOG's lexical syntax. A separator character is not always necessary in order for micro-PROLOG to determine the end of one syntactic object and the beginning of another. On the other hand, insertion of a separator never does any harm.

## 2.5 Lists

Lists are the structured objects of micro-PROLOG. The simplest list is the empty list

()

which is written as a left (round) bracket followed by a right (round) bracket (separated by any number of spaces, including none). If there are terms inside the brackets we have a non-empty list. Thus

(2 3 5)

is a list of the three integers 2, 3, 5. The spaces between the integers are important. The list

(235)

is the list of one integer 235.

The objects in a list can be any term, including another list. For example,

((2 3) apple x (-2.3 &&))

is a list of four objects:

a sublist (2 3) of the two integers 2, 3

the constant apple

the variable x

a sublist of two objects: the floating point number -2.3 and the graphic constant &&.

As in LISP, sublists can be nested to arbitrary depth, e.g.

((a (b c)) ((d)) ((e) f) h))

is a list of three sublists.

### 2.5.1 The list constructor

Internally, non-empty lists are represented as structures built up from the empty list by a sequence of list constructors. The micro-PROLOG list constructor analogous to the LISP dot. For example,

(a b c)

is represented internally as

(al(b)(c|()))).

The inner-most expression (c|()) represents the list constructed from the empty list by adding c as a front element. That is, (c|()) is the internal representation of the single element list (c). So (b|(c|())) is the list (c) with b added as a front element. In other words, it is the internal representation of the list (b c). Finally,

(al(b)(c|c|()))

is the list (b c) with a added as a front element, which is the list (a b c).

The notation (t1 t2.....tn) for a list of n terms is just an accepted abbreviation for the explicitly constructed list



```
(t1|(t2|....(tn|())....))
```

More generally,

```
(t1 t2....tn|t)
```

is an accepted abbreviation for the explicit construction

```
(t1|(t2|....!(tn|t)....))
```

when  $t$ , as well as each of  $t1 \dots tn$ , is any term - i.e. a number, constant, variable or list. The  $!$  should be read as: followed by. The above term is the list of the  $k$  elements  $t1, t2, \dots, tk$  followed by  $t$ .

#### examples

```
(a b c|x)
```

is the list comprising the sequence  $a, b, c$  followed by  $x$ . A list term of the form

```
(t1|t2)
```

is the list comprising the term  $t1$  followed by  $t2$ . In LISP parlance,  $t1$  is the head (or CAR) of the list and  $t2$  is the tail (or CDR) of the list.

#### 2.5.2 List patterns

Any list term with variables is a list pattern. The above

```
(a b c|x)
```

is a list pattern representing any list that begins with the constants  $a, b, c$  in that order. Since  $x$  may be the empty list, the list  $(a b c)$  is covered by this pattern. List patterns have a crucial role in micro-PROLOG. Relations over lists are defined using list patterns. Further examples of list patterns are:

```
(i) (x|y)
```

represents any list comprising at least one element  $x$ . Again, since  $y$  can be the empty list, this includes the single element list  $(x)$ . Note that  $(x)$  is also a list pattern, representing any list of exactly one element.

```
(ii) ((x|y)|z)
```

represents any list that starts with a non-empty list (the pattern  $(x|y)$ ). Since  $z$  and  $y$  can both be the empty list, the list pattern  $((x))$ , denoting a singleton list with a singleton list as its only element, is a special case of the pattern  $((x|y)|z)$ .

```
(iii) (x1 x2 (x3 x4))
```

is a list of three elements, whose third element is a list of two elements.

```
(iv) (x1 x2|y)
```

is a list of at least two elements  $x1 x2$ , in that order. It covers the case of a list of exactly two elements since  $y$  may be  $()$ . In each of the above patterns the variables represent any term, including lists. There are no types in micro-PROLOG over and above types of list structures represented by list patterns. Variables always represent any term.

```
(v) (x1 x2|(x3 x4))
```

is the list of two elements  $x1, x2$  followed by the list of two elements  $x3, x4$ . In other words, it is the list

```
(x1 x2 x3 x4)
```

which is an equivalent pattern. Both are represented internally as

```
(x1|(x2|(x3|(x4|()))))
```

Although the bar is the only data constructor in micro-PROLOG, which is a major difference between it and other PROLOGs, other constructors can easily be simulated by using list notation. In most PROLOGs, any constant  $f$  can be introduced as a data constructor with  $f$ -structures denoted as terms of the form

```
f(t1,...,tn)
```

In micro-PROLOG, you can represent this, as in LISP, as the list

```
(f t#1 ... t#n)
```

where  $t\#i$  is the list representing the term  $t_i$  and the terms are separated by spaces not commas.

All micro-PROLOG constructs are expressed in terms of the four types of term discussed so far: Numbers, Constants, Variables and Lists. The higher level syntactic constructs such as atoms and clauses make use of these basic objects, and they are generally list structures.

#### 2.6 Atoms

Atomic formulae, or atoms, are the primitive sentence forms out of which program statements and queries are constructed. In the usual syntax of predicate logic an atomic formulae is an expression of the form

```
R(t1,...,tn)
```

where  $R$  is a relation name (more formally, predicate symbol) and  $t1, \dots, tn$  are its argument terms. In micro-PROLOG, the atomic formula is written as the list

```
(R t1 .. tn)
```

comprising the relation name followed by the sequence of

## 2. Syntax of micro-PROLOG

arguments. There are no separating commas. Spaces are used if needed (see section 2.10).

### Example

```
(father-of tom bill)
```

is the atom which expresses the relation of father-of between tom and bill.

The relation name of an atom, that is the head of the atom list, must be a constant. (But see section 2.9).

### 2.7 Clauses

All PROLOG program statements correspond to a restricted form of sentence of predicate logic called **Horn implications**, or **definite clauses**. These are sentences of the general form

```
<atom> if <atom1> and <atom2> and...and <atomk>, k ≥ 0
```

in which each variable appearing in the sentence represents any term. In the micro-PROLOG basic syntax the logical connectives if and and are dropped, and the clause becomes the list of atoms

```
(<atom> <atom1>...<atomk>)
```

Since each atom is itself a list, a clause is a list of sublists in which each sublist is a list that begins with a constant which is a relation name. (But see section 2.9 for exceptions.) The first atom is the **head** of the clause, the remaining atoms comprise the **body** of the clause.

### Examples

```
(i) ((father-of tom bill))
```

is the unconditional statement that the atom

```
(father-of tom bill)
```

represents a true fact.

```
(ii) ((App () x x))
```

is the unconditional statement that for all x

```
(App () x x)
```

is an instance of the Append relation.

```
(iii) ((App (x|y) z (x|y1)) (App y z y1))
```

is the conditional statement that for all x, y, z and y1

```
(App (x|y) z (x|y1))
```

is an instance of the App relation if

```
(App x y y1)
```

## 2. Syntax of micro-PROLOG

is an instance of the App relation. These are both true statements when (App x y z) is understood to mean that z is the result of appending (concatenating) a list x to the front of a list y. Note the role of the patterns (x|y) and (x|y1) in the second clause. They tell us that the clause deals with the case of a non-empty front list and the repeated x in the two patterns tells us that the first element of the front list becomes the first element of the concatenation.

### 2.8 Comments

Since micro-PROLOG is first and foremost an interactive system comments in the form of text in a source file that is ignored on loading is not supported. However, comments can be inserted in a program as statements about some comment relation, or as comment conditions in a clause.

#### 2.8.1 Comment conditions

The built-in relation /\* is a multi-argument relation that is always true. Thus an atom with /\* as relation name, followed by any term or sequence of terms, is always skipped over by the micro-PROLOG interpreter.

#### Example

```
((App () x x)
  (/* App is the concatenation relation for lists))
```

is a commented assertion about the App relation. It will be slightly slower in use than the uncommented assertion.

#### 2.8.2 Comment clauses

The relation name used for comment clauses can be chosen by the programmer, however it cannot be /\*. No clauses are allowed that make statements about the primitive relations of micro-PROLOG. An attempt to enter such a clause results in an error.

Suppose the programmer uses the relation name comment. The clause

```
((comment App (the concatenation relation for lists))
```

adds the same information about the App relation. The main difference is that it is not embedded in the App program and so is not automatically shown in a listing of App program. The comment for App must be retrieved by either listing the comment relation or by querying the comment relation. We deal with listing and querying in the next chapter.



## 2.9 Meta-variable

The reader may skip this section on a first reading of the manual.

As an extension to the clausal syntax described above variables can appear in certain positions in a clause to name 'meta-level' components of the clause.

A variable can be used in place of the predicate symbol of an atom in the body, it can name a whole atom in the body, or it can be used to name the 'rest' of the body. These various uses of variables in the bodies of clauses are called the 'meta-variable' facility. This is to indicate that at run-time the variables concerned will be bound to terms which become the relevant components of the clause.

The meta-variable is very important to the usability of micro-PROLOG, it enables many of the second order programs found in LISP (say) to be expressed succinctly in micro-PROLOG.

When the micro-PROLOG interpreter makes use of a clause containing a meta-variable the variable must have been bound by the time that the meta-variable condition is reached. Moreover the term binding must be a valid syntactic item for the position of the meta-variable in the clause.

So, if the meta variable replaces a relation name, it must have been bound to a constant; if it replaces an atom it must have been bound to a list that is an atom; if it replaces the remainder of the body, it must have been bound to a list of atoms. If this is not the case, an error is signalled.

## 2.9.1 Meta-variable as Predicate Symbol

A variable can be used as the predicate symbol of an atom in the body of a clause. (The head atom of a clause must always have a constant as the predicate symbol. This names the relation that the clause is about.) A predicate symbol meta-variable must be bound to a constant before the atom is evaluated. The constant is taken as the predicate symbol of the atom for this call.

This form of the meta-variable can be used to implement the equivalent of the MAP functions in LISP. It is also similar to the facilities to pass procedures as parameters commonly found in more conventional programming languages like Pascal, ALGOL etc.

In the example program below Apply applies a test to each of the elements of its list argument. A call to Apply takes the form: (Apply OK <list>) and it succeeds if (OK Y) is true of each element Y of <list>. The X argument, the relation to apply to each element of the list, must be given in any call to Apply.

```
((Apply x ()))
((Apply x (Y|Y))
 (x Y)
 (Apply x Y))
```

## 2.9.2 Meta-variable as an atom

A variable can also be used instead of an atom in the body of a clause. In this case the variable must be bound to a term which names an atom when the variable is 'called'.

This variant of the meta-variable is used to implement some of the meta-level extensions to the language. For example, the

following program 'evaluates' a list as though it were a list of atoms:

```
((Eval ()))
((Eval (x|X))
 x
 (Eval X))
```

This use of the meta-variable does not have a direct counterpart in Pascal, it would correspond to passing an expression as a parameter to a procedure; the closest comparison is with the call-by-name mechanism in ALGOL [Naur 1962].

## 2.9.3 Meta-variable as the tail of the body of a clause

The final variant of the meta-variable is its use as the tail of the body of a clause or as the entire body of a clause. A variable in this case represents a list of procedure calls, rather than just a single call. For example, the following program encodes the disjunctive operator OR (available as a built-in program):

```
((OR x y) | x)
((OR x y) | y)
```

The use of the bar in these two clauses implies that the variables x & y name lists of atoms, and during execution they must be bound to lists of the correct format. Each list is interpreted as the body of the clause.

Again, this use of the meta-variable has a loose counterpart in conventional programming languages; in particular the closest comparison is with the label parameter passing mechanism of ALGOL 60, the replacement body is 'jumped to' rather than being called as with the meta-variable as atom.

## 2.10 Lexical syntax

In this section we describe in more detail the lexical syntax that micro-PROLOG uses; the section can be omitted on a first reading of the manual.

The lexical syntax determines how the sequence of characters input to micro-PROLOG (either from the console or from a disk file) are grouped together into tokens.

In some sense the notion of token is a generalisation of word; in that tokens form the smallest groups of characters that can have a meaning associated with them; for example numbers, names and special symbols like (, | and ) are all tokens. The lexical rules themselves however, do not attach meaning to tokens, they merely define what tokens are. In micro-PROLOG there are five different types of token: special tokens, numeric tokens, alpha-numeric tokens, graphic tokens, and quoted tokens.

## 2.10.1 Token boundaries

The boundaries between tokens are determined by separator characters and by certain changes in token type. For example, a number token can be immediately followed by a graphic token since they are of different type, however two successive number tokens must be separated by at least one separator character. The

separator characters are space, carriage return, line feed and tab. Apart from their role as token separators, separator characters are ignored on input (but see quoted tokens below).

### 2.10.2 Special tokens

Special tokens consist of single characters, called special characters. Three special tokens form part of the list syntax of micro-PROLOG:

```
( ) |
```

while the others are always regarded as single character constants by micro-PROLOG (even though they are also bracket characters):

```
[ ] < > { }
```

The use of one of these special characters always marks a token boundary. They do not need to be preceded or followed by any separator.

### 2.10.3 Alpha-numeric tokens

Alpha-numeric tokens are defined in a similar way to identifiers in normal programming languages. They consist of a letter (lower or upper case letter) followed by a sequence of letters, digits. The sign character can also be used in alpha-numeric tokens to aid readability. Some example alpha-numeric tokens are:

```
A A1 x AlB3fred father-of A-1 -A
```

An alpha-numeric token must be separated from a preceding alpha-numeric token and from a following number token by at least one separator.

### 2.10.4 Number tokens

Numeric tokens are tokens which denote integers or floating point numbers. They are described in section 2.2.

A numeric token must be separated from a preceding number or alpha-numeric token and from a following number token.

### 2.10.5 Graphic tokens

Graphic tokens are names which are built up from the non alpha-numeric (and non special) characters. The sign character can also appear in graphic tokens. This includes such characters as : = % etc. Some example graphic tokens are:

```
- = :: ?? /+ '-$ &
```

Graphic tokens need only be separated from preceding or following graphic tokens.

### 2.10.6 Quoted tokens

The final kind of token is the quoted token. This is used when the lexical roles of characters need to be ignored; it allows arbitrary characters to be grouped together as a single token. A quoted token is as a quote character " followed by an arbitrary sequence of characters (excepting the quote character itself) and terminated by another quote character. The quote character can be inserted in the token by prefixing it with the escape character @. (See section 2.3.)

Quoted tokens do not need to be preceded or followed by a separator.

### 2.10.7 The lexical rules

The parser in micro-PROLOG has the fairly simple task of recognising tokens, converting tokens into variables, numbers and constants and constructing lists out of the token sequences that begin with ( and end with ).

There is a fairly close correspondence between the lexical token types and the distinctions the parser needs to make: numbers are made from numeric tokens, graphic tokens and quoted tokens form constants.

Numbers Constants

```
( 1 2 ( $ "A string" ) )
```

Special tokens

The remaining kind of token: the alpha-numeric token is recognised either as a variable or as a constant by the parser. micro-PROLOG recognises variables by examining the first character of alpha-numeric tokens (called the prefix character). If this character is a variable prefix character and the rest of the token is made up of digits then the token is read as a variable, otherwise it is taken to be a constant:

Alpha-token

```
x rest of token
```

Prefix digits  
char

In standard micro-PROLOG the variable prefix characters are x, y, z, X, Y and Z. Appendix C gives details of how the set of variable prefix characters can be modified. Finally, when the parser encounters a ( it constructs a list out of the sequence of terms parsed from the following token sequence that terminates with the corresponding ).

### 3. Interacting with the micro-PROLOG system

micro-PROLOG is an interactive system. The top level interaction with the user is controlled through a special built-in micro-PROLOG program called the supervisor.

The micro-PROLOG supervisor provides a simple operating environment for the user. It allows programs to be entered, executed, edited, saved and loaded on disk files. The supervisor also includes the definitions for all the primitive relations that are implemented as micro-PROLOG programs. In this chapter we describe the user interface to the supervisor.

The supervisor is an integral part of the micro-PROLOG interpreter, it is always present and cannot be deleted.

#### 3.1 micro-PROLOG distribution disk

The micro-PROLOG 3.0 distribution disk contains the following 16 files:

PROLOG.COM	PROLOG.SYM
TRACE.LOG	SPYTRACE.LOG
EDITOR.LOG	MODULES.LOG
EXREL.LOG	
SIMPLE.LOG	DEFTRAP.LOG
EXPTAN.LOG	TOLD.LOG
SIMTRACE.LOG	
MICRO.LOG	ERRTRAP.LOG
UP212T03.LOG	PRODUCT.LOG

PROLOG.COM is the PROLOG interpreter with the built-in micro-PROLOG supervisor program. The file PROLOG.SYM is only needed for those extending micro-PROLOG with primitives written in assembler (see Chapter 8). All the other files, the .LOG files contain micro-PROLOG programs wrapped up as modules. They provide optional loaded extensions to the built-in supervisor. Their facilities are described in various sections of this manual.

The facilities of the first five are described in Chapter 4, the next five are described in Chapter 5, the next two (MICRO.LOG and ERRTRAP.LOG) are the subject of Chapter 6 and the last two are programs that provide compatibility with earlier versions of micro-PROLOG. PRODUCT.LOG is described in Chapter 5 and UP212T03.LOG is only of interest if you have upgraded from version 2.12 of micro-PROLOG to version 3. If you have, you will have been supplied with a separate documentation sheet describing how to use the program to update your micro-PROLOG software for version 3.

The ability to create and use modules is a distinctive feature of micro-PROLOG. Very few PROLOG systems support

### 3. Interacting with the micro-PROLOG system

modules. When modules are loaded the micro-PROLOG programs they contain do not enter the normal user program workspace (although they do reduce the size of this workspace). This is because the relations defined in a module do not enter the workspace dictionary.

Each module has its own private dictionary and a relation defined in a module can only be used by a workspace program if the module exports its name. The exported relation names of a loaded module appear to a workspace program as extra primitive relations of micro-PROLOG. Modules are described in 7.11.

The program file MODULES.LOG, described in the next chapter, can be used to create and then edit modules. When you have developed a micro-PROLOG program, and you want to isolate all but a few main relations that will be used in other programs, you should wrap up the program as a module.

#### 3.1.1 Invoking micro-PROLOG

To enter the micro-PROLOG system:

1. Insert the micro-PROLOG distribution disk (or the back-up copy disk) into a suitable disk drive (the B drive say).
2. Log in to the disk (not strictly necessary but it simplifies the loading and saving of programs from micro-PROLOG).
3. Type the command PROLOG.

For example, if your computer has two disks: the A disk and the B disk, to execute micro-PROLOG from the B disk, with the system disk in A, type:

```
A>b:
B>prolog
```

When micro-PROLOG is started up the following banner should appear at the console:

```
micro-PROLOG r.vv S/N *****
(C) 1982 Logic Programming Associates Ltd.
99999 Bytes Free
&.
```

The first two lines form the micro-PROLOG banner, and give details of the release (r) and version (vv) number. The message "99999 BYTES FREE" indicates how much memory is allocated for the work space.

The allocation is divided into two fixed areas: approximately 19% of the available memory is allocated to the storage of text for the dictionary (where the names of constants are stored), and the rest forms the heap and stack space. This latter region is where the user programs are stored, and where evaluation of programs takes place.

There is no fixed division between the stack and the heap. They grow towards each other and garbage collection of the heap occurs when they are about to collide. When garbage collection does not create enough space to continue, you get a "No space left" error message and the current evaluation is aborted with a

### 3. Interacting with the micro-PROLOG system

return to the supervisor.

Because of the trade-off between the heap and stack you can create space for an evaluation by deleting all unnecessary relation definitions and modules before the evaluation. Thus, utility modules that have been used to help develop and/or edit the program can be deleted to make room for the evaluation. This ability to load, then kill, then re-load support software as required is a significant feature of the program development environment of the micro-PROLOG system.

The last line of the banner starts with a &.. This is the system level prompt which is output by the supervisor. It indicates that the supervisor is waiting for input from the console keyboard.

micro-PROLOG commands can also be given in the initial CP/M command line. Any characters that are typed on the command line after the word prolog are taken to be the initial input to the micro-PROLOG system (i.e. without waiting for the prompt). For example,

```
B>prolog load simple
```

is equivalent to:

```
B>prolog
micro-PROLOG r.vv S/N *****
(C) 1982 Logic Programming Associates Ltd.
99999 Bytes Free
&LOAD SIMPLE
```

CP/M automatically converts characters appearing in the command line to upper case before passing them on to micro-PROLOG.

#### 3.2 Keyboard control and line editor

When reading from the keyboard the system prompts the user for input with a . prompt and then backspaces the cursor over the . (the top-level &. prompt that micro-PROLOG gives you on entry is made up of the workspace dictionary name & followed by the . read prompt). The positioning of the cursor over the . means that on certain terminals the . will be not be seen. Of course, the first character typed always erases the . prompt.

As characters are typed in response to the read prompt they are stored in a special keyboard buffer and are only 'read' by micro-PROLOG after the RETURN or ENTER is pressed. Until you hit RETURN, you can edit the sequence of characters using the line editor described below. You are actually in the input mode of this editor as you are typing in the characters up to the point that you hit RETURN.

The keyboard buffer holds up to 250 characters, so a 'line' of up to 250 characters can be edited by the line editor. Of course, all these characters will not appear on the same console display line. When the display reaches the end of a line, the display should automatically move to a new display line without your having to hit RETURN.

The cursor backspace of the edit mode of the line editor will move back over different display lines. So, only hit the RETURN when you are confident that you do not want to edit the sequence of characters you have typed in response to the read prompt. Hitting RETURN makes that sequence an entered line which

### 3. Interacting with the micro-PROLOG system

cannot be edited.

Note: the automatic transition from one display line to another is a function of the terminal/computer. micro-PROLOG has no knowledge of the transition - it is not a token boundary (see 2.10). If the token that finishes at the right end of one display line needs to be separated from the one that starts the next line you need to type a space in front of the token on the second line.

The following control keys have special significance in the input mode of the line editor:

- <Backspace> or <Rubout> will delete the last character typed in
- <Return> exits the editor and passes the current line the micro-PROLOG echoes a \$ and enters **edit mode** (see below)
- <Control-P> toggles the printer (as in CP/M)

Certain other control keys provided by CP/M are not supported by this line editor; these include:

- <Control-R> Review the line
- <Control-X> Cancel input
- <Control-U> Same

#### 3.2.1 Edit mode

You enter the edit mode from the input entry mode by hitting <escape>. In the edit mode all commands are single letters. These letters can be in either upper or lower case and are never echoed to the screen. The edit commands provide fairly simple character level editing functions such as cursor movement, character substitution, character search etc.

In the descriptions of the commands below we shall talk about a 'cursor'. This is similar in principle to the cursor on a screen, except that since the line editor is one dimensional the cursor can only move to the left or to the right. The cursor can only be 'over' an existing character in the keyboard buffer. Any attempt to move it outside existing text will cause the bell to be sounded on the terminal.

Similarly, if a character is typed as an edit command which is not recognised, or is illegal for some reason the bell is sounded on the console, and the command ignored. The edit commands are summarised as follows:

- i Insert mode (start accepting characters)
- <Return> Exits the line editor after displaying rest of line. Edited line passed to micro-PROLOG.
- <space> Move 'cursor' one character to the right. The character is echoed to the screen. If already at the end of the line then the bell is sounded instead.
- <backspace> or <Rubout> Move the 'cursor' left one

### 3. Interacting with the micro-PROLOG sys

character. A backspace is echoed to the screen. If already at the start of the line then the bell is sounded. Note that unlike in input mode the backspace does not delete the char under the cursor.

Searches the keyboard buffer rightwards from the current position for <char>. Text between the cursor and the target is displayed on the screen. If the <char> is not found then the bell is sounded and the cursor is left at the right end of the line.

Replaces the character under the cursor with <char>.

Deletes the character under the cursor. Positions occupied by deleted characters are indicated by a \$ which is now skipped over by the <space> and <Rubout> commands.

Similar to search, except that the characters between the cursor and the target are deleted. As with delete, the positions of deleted characters are indicated by \$'s.

Lists the entire line and positions the cursor at the beginning of the line. \$'s marking deleted characters are removed by this command.

Toggles the print mode; analogous to the Control-P key when in insert mode. (A p entered in edit mode will cancel a <Control-P> entered in insert mode).

This is used to extend the line. The rest of the line is displayed and insert mode is entered at the end of the current text.

This cancels the rest of the line from the cursor position and enters input mode. Useful when retyping a whole line.

Any number of terms can be typed on a line, and a term can be spread over many lines. Any excess terms (terms are read one at a time) are saved in the buffer until the next console read is executed, in which case the next term in the buffer is read and no prompt is displayed. However, excess terms in the input

### 3. Interacting with the micro-PROLOG system

buffer are discarded if there is a reported error (other than "Syntax error") before they are read. Errors cause the input buffer to be flushed.

#### 3.2.2 Type ahead

Even before an input prompt is displayed, i.e. before an input request is given, you can enter up to 16 characters by typing ahead. The characters will not be echoed until the input request is given. If more than 16 characters are typed ahead then the bell on the terminal will be sounded, if it has one. All characters after the first 16 are lost.

#### 3.2.3 Multi line terms and right bracket prompts

Constants, variable names and numbers cannot be split across entered lines because <return> is a separator. List terms can be split over more than one entered line.

If you hit ENTER before the list term is complete you will get a prompt of the form n., where n is the number of right brackets needed to complete the list. For example, we might have entered the App clause

```
((App () x x))
```

over four entered lines:

```
&(( <return>
2.App( <return>
3.)x x <return>
2.) <return>
&.
```

^ (cursor position)

The 2, 3 and 2 on the second, third and fourth lines are the bracket count components of the read prompts.

The . of these three lines will actually not be seen when the text is entered. It will have been erased by the first character typed. However, we shall always show the . in our example interactions.

The bracket count prompt is very useful when entering lists, in particular, clauses. If you are unsure about the number of closing right brackets you need, and you are sure that you do not want to do any editing of the current line, enter a <return>. The prompt will tell you how many right brackets you need to finish of the list.

### 3.3 Supervisor commands

From the user's point of view the supervisor consists of a set of commands. Commands are typed in when the supervisor has issued a &. prompt. Commands can also be invoked from program clauses by using the command name as a unary relation. Furthermore, the supervisor allows you to define and invoke your own commands because any user defined unary relation can also be used as a command. In this section we describe the various commands available. The general format of supervisor commands is:

### 3. Interacting with the micro-PROLOG system

&.<Command name><Command argument>

where the command name is a constant, and the command argument is a term whose exact form is dependent on the actual command. The command argument is what would be the single argument if the command name were used as a unary relation.

#### 3.3.1 Entering clauses

There is one exception to the command format described above. To enter a clause you simply type the clause. The clause is added to the program at the end of the sequence of clauses currently defining the relation of the head of the clause. For example:

```
&(((Parent Mary John))
&((Parent Peter John))
&((App() x x))
&((App(x) Y(x) Z))
1.(App X Y Z))
&.
```

There is no supervisor command to enter a clause at a position other than at the end of the current sequence of clauses for its relation. To do this, you have to use the primitive ADDCL relation in a query command (see section 3.3.3 below), or the structure editor (described in Chapter 4), or the MICRO extension to the supervisor (Chapter 6).

#### 3.3.2 The LIST command

The PROLOG program currently in the workspace can be listed at the console with the LIST command. When a program is listed it is displayed in an indented format to aid readability of the program.

LIST ALL

lists the entire program.

LIST <relation name>

lists all the clauses (in the current program) for the named relation. Finally,

LIST (<relation name1> ... <relation namek>)

lists the programs for each of a list of named relations.

```
&.LIST ALL
((App () X X))
((App (X) Y) Z (X) X))
((App Y Z x))
((Parent Mary John))
((Parent Peter John))
&.LIST Parent
((Parent Mary John))
((Parent Peter John))
&.
```

### 3. Interacting with the micro-PROLOG system

The variables that appear in a listed clause will usually be different from the ones you used when you entered the clause. This is because, as we noted in 2.4, variables are converted into a special internal form when a term such as a clause is read in. The name of the variable is lost, only its positions within the term are remembered. When the term is listed, each variable is given a print name from the sequence of names X, Y, Z, x, y, z, X1, ..etc. The first variable in the term is displayed as X, the next as Y and so on. That is why the second App clause is listed as above.

#### 3.3.3 The query command 2

micro-PROLOG programs can be queried using the query command ?. This takes as its single argument a list of atoms which represents a conjunction of conditions to be solved. It can also be viewed as a sequence of goals or procedure calls each of which invokes the (generally) non-deterministic program for its relation.

If a solution to the ? query is found then the supervisor displays its normal prompt:

```
&?((Parent x1 x))
&.
```

The query to show that someone x1 is the parent of someone x succeeds.

If a solution cannot be found, i.e. the query fails, then a ? is displayed before the next prompt:

```
&?((Parent x1 x2)(Parent x2 x1))
?
&.
```

The query to show that someone x1 is the parent of someone x2 and that the found x2 is also the parent of the found x1 has failed. micro-PROLOG does not automatically print any response if the evaluation of the ? query succeeds. If you want to see the values for variables for the found solution you must add an extra PP condition/call to the list. For example, to find the name of a Parent of John use the query:

```
&?((Parent x John)(PP The parent of John is x))
The parent of John is Mary
&.
```

PP is a built-in multi-argument relation that always succeeds with the side-effect of displaying its sequence of arguments at the terminal.

In Chapters 5 and 6 other forms of query are described that automatically display the solutions.

#### 3.3.3.2 Interrupts

To interrupt the execution of a goal two interrupt keys are provided. If the Control-S key is pressed, then execution is suspended. Execution continues as soon as Control-S is typed again.

To break into an execution the Control-C key is used. When a break is detected the "Break!" error is signalled and the

### 3. Interacting with the micro-PROLOG system

execution is interrupted. Appendix A gives a complete list of the signalled error conditions of micro-PROLOG. It also describes how a signalled error can be trapped by a program for the reserved relation name ?ERROR?. It gives an example definition of the relation which provides a program that displays a short description of the error together with the call being evaluated when the error occurred.

You should always have some definition for ?ERROR? present. The ERRTRAPLOG file contains a module that exports a definition for this relation. It provides a sophisticated error trap and recovery utility. Its facilities are described in the MICRO chapter, Chapter 6. However, you can load and use it separately from the MICRO system.

#### 3.3.3.2 Other uses of the query command

The primitive relation ADDCL can be used in a query command to add a clause at a specified position in the current sequence of clauses for its relation.

Thus,

```
?((ADDCL ((Parent John James)) 1))
```

will add the clause

```
((Parent John James))
```

after the current first clause for the Parent relation. Since we already have

```
((Parent Mary John))
((Parent Peter John))
```

the new clause will be inserted between these two clauses.

Individual clauses can be deleted using DELCL. This has a unary form and a binary form. In the unary form the single argument is the clause to be deleted. In this form it can be used as a command, e.g.

```
DELCL ((Parent Peter John))
```

In the binary form its arguments are a relation name and the position of the clause to be deleted. In this form it must be used inside a query command, e.g.

```
?((DELCL Parent 2))
```

deletes the current second clause for the Parent relation. See Chapter 7 for a complete description of the ADDCL and DELCL relations. See also Chapter 4 for an explanation of how to add and delete and reposition clauses using the structure editor.

#### 3.3.4 Unary relations as commands

The supervisor also provides an alternative way of querying (calling) unary relations. In the special case where the query list comprises a single atom, and that atom is unary (i.e. only one argument) then the query can be re-expressed as a command comprising the relation name followed by its single argument.



&.<relation name> <argument>

This allows you to define new 'commands' to the system. For example, suppose that we have added the clause:

```
((Show x) (? x)(PP x))
```

which uses the supervisor query command ? as a unary relation. By typing:

```
&.Show (<atom1> <atom2> .. <atomk>)
```

the Show program is invoked as though it was called with query:

```
?((Show (<atom1> <atom2> .. <atomk>)))
```

It evaluates the list of atoms and then prints the list in its solved form, i.e. with variables replaced by their answer bindings.

The ? command that we saw above is itself defined as a unary relation by the clause:

```
((? X)IX)
```

This clause uses the meta-variable feature described in 2.9 in which the body of the clause is replaced by the value of the variable X. All other supervisor commands, such as LIST, are themselves just built-in micro-PROLOG programs for unary relations that are invoked using this facility. This is why any supervisor command can be used as a unary relation in the condition of a clause of query.

### 3.3.5 Multi-argument commands

Sometimes it is convenient to define a command which has several parameters. One way to do this is to define it as a unary relation that takes a list of its parameters as its single argument. ? is rather like this. Its single argument is a list of any number of conditions to solve. Alternatively, one can define the command as a program for a unary relation that immediately reads in its extra arguments.

As an example,

```
((addcl X)
 (R Y)
 (ADDCL Y X))
```

is a program for a relation that can be used as though it were a two argument command. An example use is:

```
&.addcl 1 ((Parent John James))
```

When executed, the first argument, the 1, is read in by the supervisor and becomes the single argument to the addcl call. This invokes the single clause for the relation which immediately reads in the next term (the clause to add) using the primitive R relation. Then the two argument ADDCL is called with the position X and the clause Y. Compare this with the direct use of ADDCL using ? that we gave earlier.

## 3. Interacting with the micro-PROLOG system

### 3.3.6 LOAD and SAVE commands

A user program can be saved onto a disk file, and subsequently loaded back into the workspace. The two supervisor commands SAVE and LOAD respectively save and load the user's programs.

The formats of the load and save commands are:

```
LOAD <file name>
SAVE <file name>
```

The <file name> must be different from that of any program relation and different from the name of any currently loaded module. If it is not you will get a signalled error on trying to obey the command. If you get an error during a LOAD you should CLOSE the file with a

```
CLOSE <file name>
```

command because the file which was automatically opened for the LOAD will have been left open.

#### 3.3.6.1 File names

A file name is a constant which describes a file in the CP/M style. The general format of a CP/M file descriptor is:

```
<Drive Letter>:<File name>.<File type>
```

The drive letter and file type are optional, in which case the colon and dot (respectively) are omitted. micro-PROLOG uses the file type LOG as the default file type if one is not given, and it uses the current 'logged-in' disk drive if a drive is not specifically specified. Since the colon and the dot are not alphabetic characters a file name using them must be written inside string quotes. Some example constants describing files are:

```
"A:TRACE.LOG"
TRACE
"TRACE.LOG"
"OTHER.ASM"
"B:OTHER"
```

The quotes are needed for any file name that includes a : or lower case letters. The name as given is passed on to CP/M which treats all letters as though they were uppercase. But beware the problem mentioned above. Your file name as given in the micro-PROLOG LOAD or SAVE command must be different from that of any relation or module name. Using only uppercase letters in a file name helps to avoid that problem.

The LOAD command reads a program from the file specified and, if the program is not a module, it adds the program into the user's workspace as if it were typed in. Any program already in the system is not disturbed in any way by the load: each new clause is added to the end of the current sequence of clauses for its relation. In this way the programmer can have a library of programs, on a number of different files, which are loaded in as required when building up a new program.



The SAVE command saves the program currently in the workspace into the named file. The entire workspace is saved, i.e. all the clauses that are listed in response to a LIST ALL command. If the program is saved onto an already existing file, then the old file is renamed with file extension .BAK. This automatically ensures that at least one back-up copy of a file is maintained. A subsequent LOAD of the file containing the saved program will add the programs to the workspace.

To save the programs for named relations only you need to use the two argument form of SAVE in a query command.

```
?(SAVE FAMILY (Parent Male Female)))
```

will save all the clauses for the relations Parent, Male and Female in the file FAMILY.LOG of the currently logged in drive. The second argument to SAVE is a list of the relations to be saved.

SAVE does not delete any program from the workspace. To do that you need to use DELCL or the KILL command described below. You can change the currently logged in drive or change a disk in a drive, using the command

```
LOGIN <letter>
```

where <letter> identifies the drive (i.e. it is A or B or C etc.). Before you execute the command, put any new disk in the selected drive. Thus, to change the disk in B, put in a new disk and then enter

```
LOGIN B
```

Other built in relations for file handling are:

```
ERA for erasing files
REN for renaming files
DIR for inspecting the CP/M directory of a file
```

They are described in Chapter 7. As an example of the use of DIR,

```
?((DIR * X)(PP X))
```

will give you a list of all the names of the .LOG files of the currently logged in disk.

### 3.3.7 The DICT relation

All constants that are used in a user program are stored in a dictionary and uses of the constant in the program become pointers into the dictionary. So do not worry about using long relation names. You can see this dictionary by executing

```
LIST DICT
```

What you will see is something of the form

```
((DICT & () (...) App Parent John .....))
```

which is a clause for the relation DICT which is a multi-argument relation. The first three arguments are:

- (1) the constant & which is part of top-level & prompt and is the name of the workspace area,
- (2) the empty list,
- (3) a list that will be empty unless you have loaded any modules, if you have, the second list will be all the names exported by currently loaded modules.

The remaining arguments are all the constants used in the workspace program.

The workspace name & is there because this is also the form in which you will see the dictionary of a module once the module has been opened (see 7.11). & is the name of the special root module which is the workspace area. The first argument of DICT is always the name of the current module. For any module other than the root module the second argument will not be the empty list. It will be the list of all the exported names of that module.

The garbage collector clears constants from the workspace dictionary once there is no longer a reference to them in the program.

### 3.3.8 The KILL command

The KILL command can be used to delete all the clauses for a particular relation or list of relations.

```
KILL Parent
```

deletes all clauses for Parent and

```
KILL (Parent App)
```

deletes all clauses for Parent and App.

```
KILL ALL
```

will delete all clauses in the current workspace. It should be used with care, and usually only after the use of SAVE. A final form of use of KILL is

```
KILL <module name>
```

This gets rid of a loaded module.

### 3.3.9 Exiting the system

To exit the micro-PROLOG system to CP/M use the command:

```
QT.
A>
```

The following the QT is needed to give an (ignored) argument to QT. (Remember all commands must have one argument.) It can be any argument:

```
QT goodbye
```

will work as well but is not as brief. Notice that in this case the separating space is needed.

### 3.4 Pragmatic Considerations for Programmers

The principal limiting resource in micro-PROLOG is space. To help to conserve space micro-PROLOG incorporates a number of space saving features. To maximise their effect the programmer should be aware of them, so this section describes some of them and how they operate. Note that space saving does not affect the logic of the running program; it may only affect whether a program can run in the space available.

The features of micro-PROLOG which affect the space used by a program are:

1. **Organization.** The evaluation area in micro-PROLOG is organised as a stack and a heap. The stack contains the activation records and the locations for the variables of the evaluation. The stack grows with recursion and pops normally only on backtracking. It records the state of the evaluation of the current supervisor command. The heap contains the values of variables. It also contains the program clauses and other permanent data objects.

Periodically the stack and heap collide, at which time the heap is garbage collected. The garbage collector is actually called whenever the stack and heap grow too close to each other; the point at which this is done is automatically computed by the system depending on the available memory and the relative sizes of the stack and heap.

The garbage collector is a 'Mark and Collect' garbage collector, which means that all the free space in the heap is collected together into a list. The heap is also 'cut down' if there is free space at the end of it. This has the effect (hopefully) of leaving a clear region of memory between the stack and heap, allowing execution to continue.

If the garbage collector fails to find sufficient space then the evaluation aborts with the message "No space left".

Note that the allocation algorithm means that as memory gets tight the garbage collector gets called more and more often; this can have a dramatic effect on the performance of the system. Normally garbage collection takes a very short time (about 0.25 secs) and is not a big overhead.

2. **Success Popping.** micro-PROLOG performs special actions when a procedure call has been deterministic - when it has been solved using the last clause for its relation and the evaluation of each of the calls in the body of the clause has been deterministic. When such a deterministic call is solved, the activation record for the clause that solved the call is popped off the stack (it will always be at the top of the stack) in exactly the same way that the record of a procedure invocation is popped in a conventional recursive programming language. The activation record can be discarded because it will not be needed to determine what next clause to try in order to solve the call should there be backtracking to the call.

The alternative situation, where either the evaluation of one of the calls in the body of the clause was not deterministic or it is not the last clause, means that the activation record is left on the stack.

The expert programmer can give control information that allows a success popping even when the evaluation of the call

appears to have been non-deterministic. That is, when there are still untried clauses for the call or untried clauses for one or more calls solved during the evaluation of the call. He can do this by inserting the / backtracking control primitive in the clause (see Chapter 7).

If a / is evaluated in the body of an invoked clause it has the effect of popping from the stack any activation records left there by the evaluation of the calls that precede the / in the clause, so the evaluation of these calls is now deemed to have been deterministic. It also makes micro-PROLOG treat the clause as though it were the last clause for the call that invoked the clause. In consequence, a / at the end of a clause always results in a success popping of the stack if the use of the clause solves some call.

3. **Tail recursion** is the name given to that form of recursion which is actually equivalent to a loop. micro-PROLOG can detect this special case of recursion, and when a tail recursive call is also deterministic then micro-PROLOG does not grow the stack when entering the call.

A classic example of the power of tail recursion in saving space is during a list append. If we write the append program as:

```
((append () x x))
((append (x|x) Y (x|Z))
 (append X Y Z))
```

then for all the deterministic uses of the program the stack does not grow during the evaluation for any length of input list. The recursive definition of append is executed as though it were written as a WHILE loop.

The general conditions for a tail recursion optimisation are as follows. Consider a clause of the form

```
((R t1..tk) (A1)..(An))
```

in which (A1) .. (An) are atoms in the body of the clause. Suppose that it is invoked by some call (R t'1..t'k). If the evaluation of the calls (A1)..(An-1) is deterministic, success popping of their evaluation traces will leave the activation record corresponding to the use of the clause at the top of the stack.

Further suppose that there are no other clauses to try for the call (R...) that invoked the clause, in other words, the above is either the last clause for R or a / was executed in the body of the clause. In this situation, the activation record for the clause is not needed for backtracking on the call. (If the last but one atom in the clause is / these first two conditions will be satisfied because of the effect of its evaluation.

Finally, let us suppose that (An) is now matched with the last clause for its relation, so the activation record of this new clause is does not need to be left on the stack for possible backtracking on (An). In this circumstance, the activation record is thrown away and replaced by the activation record of the clause invoked by the last call (An).

Note that (An) need not be a call of the relation R. So the tail recursion optimisation of micro-PROLOG is a generalisation of the tail recursion that corresponds to iteration.

4. **Non-structure sharing.** micro-PROLOG is a so-called 'non-structure sharing' implementation. Briefly this means that when a variable is bound during a unification its value is explicitly computed and placed, if necessary, in the heap.

The effect of this, together with garbage collection, success popping and tail recursion, is to limit the amount of data currently in the stack and heap to that which is actually needed, though it does lead to an overall increase in memory turn-over. It also has a space benefit in that for certain simple, but common, cases the value of a variable takes occupies less space than in the more normal 'structure-sharing' implementations of PROLOG.

To take full advantage of the success popping and tail recursion space saving optimisations the programmer should try to ensure that micro-PROLOG can always detect determinism in a program. This means, for example, putting the base case of a program (such as that for append) before the general case, and using the IF condition and / where they are applicable (see Chapter 7). Programs optimised for space in this way tend to be less optimal with respect to speed of execution, and vice versa.

## 4. Utility modules

### 4.1 Tracing Execution

A trace program is provided with the system as a module with the name trace-mod in the file TRACE.LOG. The trace module allows, interactively, selective tracing of the execution of a query.

To use the trace program execute a LOAD TRACE command. This will load the module that is in the file. To get rid of the program when you have finished tracing do a KILL trace-mod. Note the asymmetry. To bring in a module you do a LOAD using the name of the file that contains it. To delete it you do a KILL using the name of the module. Nearly all the modules provided on the distribution disk have a name of the form: <name>-mod where <NAME>.LOG is the name of the file that contains them.

The module exports the relation name ???. This is used in exactly the same way as the supervisor ? query command, e.g.

```
l.??(<atom1>..<atomk>)
```

The difference is that you are lead through the evaluation of the query step-by-step, a call at a time.

When entering a call for the first time a message of the form

```
<call id>(R <seq. of args of call>)
```

is displayed at the console. The term printed represents the call/condition just before any attempt at evaluation, with all of the known values of variables substituted in place. The <call id> is a list of numbers identifying the ancestry of the call back to the original query. The length of the list corresponds to the depth of the evaluation. As an example, the list

```
(1 3 2)
```

identifies the call as the first condition of the body of a clause invoked to solve the 3rd condition of the body of a clause invoked to solve the 2nd condition of the original query.

If the relation name R of the call refers to one of the micro-PROLOG primitives (all described in Chapter 7) then the call is immediately executed. (The logical primitives IF, OR, NOT, ISALL and FORALL are an exception. These can be traced - see below.) If the call is for a user defined relation then the message trace? is also printed and the read prompt displayed. One of the allowed trace responses must then be entered.

The trace responses allow selective tracing of the program. For example, low level (or already debugged) programs can be skipped by responding: n i.e. no tracing. The allowed trace responses are:

```
n(for no), y(for yes)
```

```
s(for succeed), f(for fail),
q(for quit)
```

1. n

If n is entered the call is executed without tracing. In this case one of two things normally happens: either the call is solved or the attempt to solve it fails. If it fails then the message

```
<call id> failing (R <seq. of arguments of call>)
```

is displayed at the console, and the system backtracks. This failure may cause backtracking on calls that were previously solved if there are no untried ways of solving these calls. Each time there is backtracking on a call, the call is displayed with the failing message.

If the evaluation of a call succeeds then the message

```
<call id> solved (R <seq of instantiated arguments of call>)
```

is displayed. In this case the call is printed with the answer bindings substituted, so that you can see the result of the evaluation just completed. If the call was the last in the body of a clause then the immediate ancestor of the call is also solved, in which case a solved message is displayed for it too.

After solving a call remaining unsolved calls are entered and traced in turn. This continues until the last call of the original query is solved or until its first call is failed.

2. y The y response allows the tracing of the evaluation of the call. The trace program looks for a clause to match the call. If no clause matches

the call, the failing message described above is displayed. If there is a clause that matches (i.e. a clause whose head atom unifies with the call) the message

```
<call id> matches clause n
```

is displayed. The n is the position of the clause in the sequence of clauses for relation of the call.

If the clause that matches the call is an assertion, then since there are no atoms to trace inside the body of the clause, the call has been solved and a solved message displayed. Otherwise each of the atoms in the body of the clause are entered and traced in turn. On entering the body of any clause that matches the call the <call id> is extended by a first number that identifies the position of the call in the body of the clause. The clause being used is also identified by a message of the form

3. s Entering s allows one to arbitrarily solve a call. The immediate response is a printing of the call with the solved message. The trace program does not try to use any clauses for the call, nor does it instantiate any variables in the call.

4. f Entering f allows the user to arbitrarily fail the call, and to cause the evaluation to backtrack. The failing message for the call is displayed.

5. q Entering q quits the trace evaluation. It is used to obtain a quick exit. You are returned to the supervisor.

The trace program insists that a legal trace response is entered. If an erroneous response is input the program displays the message:

```
ENTER y n s(for succeed) f(for fail) q(for quit)
```

and prompts the user again. When you are asked whether you want tracing inside one of the logical primitives IF, OR, NOT, ISALL or FORALL the allowed responses are only y or n. But this restriction is indicated by the prompt trace?(y/n) that you will get in this case. To this prompt any response but y is taken as n.

#### Warning

The use of the trace program greatly increases space demands on the heap and stack, thus programs which run without tracing may well run out of space when traced.

As we mentioned earlier, the trace program, being a module, will not be loaded into the user program workspace. So when you do a LIST ALL you will not see any of the trace program. (However, if you do a LIST ?? you will be able to see the clauses for ?? since this is an exported relation of the module.) Loading the trace module does however reduce the amount of space available for user programs. It should therefore be deleted when you have finished using it with the KILL trace-mod command. You can always re-load it when required.

#### 4.2 Spypoint tracing

Sometimes, all that you want to trace is the entry/exit behaviour of some of your programs. A utility module called spytrace-mod enables you to spy point trace a program in this way. To use the module do a LOAD SPYTRACE. To get rid of it when you have finished, do a KILL spytrace-mod.

When you load the file you will find that an extra clause

```
((spyoints on))
```

has been added to your workspace program. This is a message that causes entry/exit information to be given for all the spy specified relations. The module exports three relations: spy, unspy, spying. The first two are unary relations that are used as commands. The spying relation has two uses, as a command to turn the tracing on and off, and as a two argument relation to give entry/exit information for a particular call.

##### 4.2.1 The spy command

If you want to have entry/exit information for all calls for a user defined relation R, do a

```
spy R
```

command. If you now LIST R you will find an extra clause has been added to the front of its program. The spy command has added the clause.

```
((R|X)
(spypoints on) {check that spy relations should be traced}
{execute / to prevent other clauses being
used to solve this call}
(spying R X)) {evaluate (R|X) call using the other
clauses giving some trace information}
```

(The "{,}" bracketed comments will not appear.) If the ((spypoints on)) clause is not in the workspace, the user provided clauses for R will be used to solve the call in the normal way. If it is present, the call (R|X) is evaluated by the call (spying R X) to the spying relation.

You cannot spy a micro-PROLOG primitive relation or any relation exported by a module. The attempt to do so will result in the "Cannot add clauses for" error being signalled as the spy command tries to add the extra clause.

#### 4.2.2 The spying relation

This takes two arguments, the name of the relation R of the call to be entry/exit traced and the list X of the arguments of the call.

The evaluation of

```
(spying R X)
```

is equivalent to the evaluation of

```
(R|X)
```

except that some trace information is given as the call is evaluated.

On entry, a message of the form:

```
<call id> : R <list X of arguments of call>
```

is displayed. The <call id> is an integer. The <call id> is increased on each call of spying and decreased if the call fails. During the evaluation the call to R it is identified in subsequent trace messages by the <call id>.

As each user supplied clause that matches the call is tried the message

```
<call id> matches clause <clause number>
```

is displayed. The <clause number> is the position of the clause in the listing of all the clauses for R including the clause added by the spy command. So the first user clause for R has clause number 2.

If the call is solved using this clause the message

```
<call id> solved R <list X of instantiated arguments>
```

is displayed giving any solution bindings for the call.

Finally, if the call fails, the message

<call id> backtracking on R <list of original arguments X> is displayed. After this message, the <call id> will be re-used to identify a different spying call.

The spying relation can be used directly in user supplied clauses and queries. For example, the query

```
?((spying father-of (tom X))(spying male (X)))
```

will give you trace information on the evaluation of the two calls even if father-of and male are not spy specified relations.

#### 4.2.3 The spying command

The command

```
spying off
```

will remove the

```
((spypoints on))
```

clause from the workspace and replace it by

```
((spypoints off))
```

It switches off the spying of all the spy specified relations.

The command

```
spying on
```

has the reverse effect.

#### 4.2.4 The unspy command

The command

```
unspy R
```

will remove the extra clause added for R by the spy command. It stops all the automatic spying of calls to R.

#### 4.3 The micro-PROLOG Structure Editor

The micro-PROLOG structure editor allows the PROLOG programmer to edit programs within the micro-PROLOG environment. It is an editor which takes into account the list structure of micro-PROLOG clauses and terms. Since the editor is itself written in micro-PROLOG it is easy to extend and modify, should the need arise.

At any moment the editor is focused on a current term which has an immediate context, which is the list that the current term is in. To act as an 'aide-memoire' the editor uses the current term to form its prompt when the editor is ready to accept a command. At the top-most level of editing a program (where the current term is one of the clauses of the program and the context is the list of clauses for a relation) the editor prefixes the prompt with a number; this number indicating the

position, within the list of clauses, of the current clause.

If at any time the current term 'pointer' is not a term or clause in the program then the editor displays **No term** or **No clause** as its prompt. The commands allow changing the current term, changing the context, moving or deleting or modifying the current term, and inserting new terms.

#### 4.3.1 Edit Commands

Before using the editor it is necessary to **LOAD** the editor module (called **editor-mod**) using the command:

**LOAD EDITOR**

To delete it when you have finished use:

**KILL editor-mod**

The editor is invoked using the command:

**EDIT <relation name>**,

for example to edit the likes program enter:

**EDIT likes**

The editor uses the first clause in the program (if the program is non-empty) as the initial current term. In the case of the likes program this could be:

[1]((likes John Mary)).

If there are no clauses for likes you will get

[0] No clause.

In which case the first thing you should do is add a clause using the **a** command. The editor offers a handy tool for building new programs as well as for modifying existing ones.

When the editor displays its prompt it is ready to accept an **edit command**, which at the top level can be any of the **i(insert)**, **a(append)**, **k(kill)**, **n(next)**, **b(back)**, **m(move)**, **c(copy)**, **t(text)**, **e(enter)** and **o(out)** commands. The edit commands are divided into two groups: those which move the current term pointer in the structure of the terms being edited, and those which change the terms in some way.

#### 4.3.2 Cursor Movement Commands

There are four commands which can be used to walk over the program; these are **n**, **b**, **e** and **o**.

1. The **n** (next) command changes the current term to the next term to the right in the immediate context. At the top level this means move to the next clause. As an example, if the current term is (A B), and the immediate context is (C (A B)(D)) then the **n** command moves the current term to (D):

(A B).n  
(D).

whereas at the top-level we get the next clause:

[1] ((likes John Mary)).n  
[2] ((likes X Y) (knows X Y) (likes Y X))

If the current term was already at the last term in the immediate context, or if it was the last clause in the program, the pointer is stepped on, but the current term becomes **No term**, (or **No clause** at the top level) indicating that it is not actually pointing to a term or clause. It is impossible to step beyond this point.

2. The **b** (back) command is the inverse of the **n** command, it is used to step back to the term to the left of the current term in the immediate context, or to step to the previous clause. To undo the effect of the previous **n** command above:

(D).b  
(A B).

[2] ((likes X Y) (knows X Y) (likes Y X)).b  
[1] ((likes John Mary)).

If the current term were already the first term, then the **b** command steps back to in front of it, again causing the prompt to become **No term** (**No clause**). e.g.

(A B).b  
C.b  
No term.

[1] ((likes John Mary)).b  
[0] No clause.

It is not possible to move before this point.

3. The **e** (enter) command steps 'into' a term or clause so as to edit its components. The term being stepped into must be a list structure. (You cannot get 'inside' a number, constant or variable using these structure move commands. To change these you must use the **t** (text) or the **s** (substitute) command.) The immediate context becomes the list term just entered, and the current term is the first element (if any) of that list. So in our example, if we enter the list (A B) we change our immediate context and point to A:

(A B).e  
A.

or

[2] ((likes X Y) (knows X Y) (likes Y X)).e  
(likes X Y).n  
(knows X Y).n  
(likes X Y).

Note what has happened as we stepped along the entered clause. As we reach each atom it is printed with the variables in the atom assigned print names from the list X, Y, Z, X, ... This means that sometimes it will appear that the editor has changed the variable names.

As we moved to the (knows X Y) atom of the clause we actually got the same names displayed by accident. When we moved to the (likes Y X) atom of the clause this is displayed as (likes X Y). Within this atom, the Y that we saw when the whole clause is displayed is the first variable, so its print name is X. The variable we now see as X is the variable that we saw as Y at the whole clause level.

This change of names as we enter and then step along a clause can be a little disconcerting. It does not matter unless you want to change a variable in a clause. To do this you should use the t text edit command at the whole clause level or at least at a term level in the clause that covers all the occurrences of the variable you want to change.

4. The o (out) command is the inverse of the e (enter) command. The current immediate context becomes the current term, and the 'old' immediate context (prior to the corresponding e command) is re-established as the immediate context. The o command is also used to exit the editor, when at the top level:

```
((likes X Y).o
[2 ]((likes X Y) (knows X Y) (likes Y X)).o
Edit of likes finished
&.
```

The o command may fail if the entered term has been incorrectly changed. This will happen if on returning to the outer level the predicate symbol of the head of the clause has been changed to a variable or the name of a primitive relation. When an edit command fails the editor responds with a ? and re-prompts at the appropriate level. If you change the relation that a clause is about to another allowed program relation name you will be asked if you want the clause added to the program for the new relation; if you respond yes, you will be asked for the position for the clause. This should be a positive integer. The clause will be inserted after the current clause at that position. With these four cursor control commands any sub-term of a program can be reached. In the next section we look at those edit commands that directly change the current term.

#### 4.3.3 Edit change Commands

There are five commands which directly affect the current term; these are i(insert) a new term, a(append) a new term, k(kill) the current term, s(substitute) another term, m(move) the term, c(copy) the term and t(text) edit the term. The c command is only available at the top level and the s command is only available below the top level.

1. The i (insert) command inserts a term before the current term. The i is followed by the term to insert as in:

```
(A B).1 (F)
(F).
```

The new term just inserted becomes the new current term, the old one can be regained by stepping on to it with the n command. At the top level the i command inserts a new clause into the program. In this case the form of the clause is checked to ensure that at least the clause is for the relation currently

being edited. If it is not, or the clause is incorrectly formed, the insert fails and the old term is re-displayed as the prompt.

```
[1] ((likes John Mary)).1((likes bill mary))
[1] ((likes Bill Mary)).n
[2] ((likes John Mary)).
```

2. The a (append) command appends a new term (or clause) after the current term. Otherwise it is like the i command.

3. The k (kill) command deletes the current term from the immediate context. The term (or clause) to the left of the deleted term in the immediate context becomes the new current term, if there is not a previous term (or clause) then the current term becomes **no term (no clause)**. We can delete a particular element of a list by using a sequence of cursor movement commands to move to the required term and then using the k command.

For example, to delete the third element of (A B C D):

```
(A B C D).e
A.n
B.n
C.k
B.o
(A B D).
```

4. The s (substitute) command replaces the current term with a new term. The argument to s is a pair:

```
(t1 t2)
```

The term t1, is unified with the current term, and then the current term is replaced by t2. The use of unification allows quite powerful pattern matching, but more importantly the specification of the replacement can make use of variables bound in this match. For example, to reverse the first two elements of a list:

```
(A B C D). s((x y|z)(y x|z))
(B A C D).
```

**Note.** The s command is **not** available at the top level.

5. The t (text) command allows the current term to be changed using the line editor described in 3.2. It displays the term on one or more lines of the console and positions the cursor at the beginning of the line. You are in the edit mode of the line editor in a state equivalent to just having executed the l command. The edit mode commands described in section 3.2 are available to modify the text of the term. Upon hitting the RETURN key the system reads the text back in and replaces the current term by it.

If you use t to modify a term which is a component of a clause be careful not to change the displayed names of any variables that also occur outside the term within the clause. If you leave the names unchanged then the link of these 'global' variables with their occurrences outside the term will be restored on exit from the text edit. This is possible because the primitive RFill relation (see Chapter 7) used by the t



command remembers the print name it assigns to each variable (which has a unique internal name) before it puts the text of the term into the keyboard buffer. When it reads in the text it replaces the occurrences of these remembered print names with the original internal names of the variables. This re-establishes the link with the variable occurrences in the rest of the clause when the reconstructed term is inserted in the clause.

6. The c (copy) command allows the copying of a clause. It can only be used at the clause level. It is very useful when building up a program in which two or more clauses differ only slightly. You insert one, then copy it using c. c places the copy after the copied clause and makes it the current clause. You then edit the copy to give the variant of the first clause.

7. The m (move) command allows you to move a term within the current context. At the top level it repositions a clause within the sequence of clauses for its relation. m takes one argument, an integer which gives the displacement to the right or left within the current context. A positive integer n moves the term to the right (forwards) n positions, a negative integer -n moves it to the left (backwards) n positions. You cannot move the term outside the current context. If you give too large a right move the term is simply placed at the right end, too large a left move puts it at the left end. The moved term remains the current term after the move.

```
[1] ((Likes Bill Mary)).m 1
[2] ((Likes Bill Mary)).b
[1] ((Likes John Mary)).n
[2] ((Likes Bill Mary)).m -4
[1] ((Likes Bill Mary)).
```

#### 4.3.4 Restructuring Lists

Given the importance of lists in micro-PROLOG, it is especially important to be able to repair an arbitrarily damaged list. Where it is just a sub-term of a list that is damaged, the above commands are sufficient. However, a problem arises if some brackets have been put in the wrong places. For example, a left bracket can be easily missed as in:

```
[1] ((Prog A X) PR X Y)
```

and a right bracket could be put in too far to the left, as in

```
[2] ((Prog A X)(PR) X Y)
```

or too far to the right, as in

```
[3] ((Prog A X (PR X Y)))
```

What this means is that the clause has the wrong sub-list structure. The editor has two simple primitives which can be used to re-structure the sublists of a clause: w(wrap) and u(unwrap).

1. The w (wrap) command takes a number of terms from the immediate context and wraps them up into a list, which becomes the current term. The w command has an argument: the number of

terms to wrap starting from the current term. If 0 (zero) is used then no terms are wrapped, i.e. the empty list () is inserted. If 1 (one) is used then the current term only is wrapped, if 2 (two) then the current plus the next term (to the right) are wrapped, and so on up to the number of the remaining terms in the immediate context.

For example, to wrap up the middle two elements of (A B C D):

```
(A B C D).e
A.n
B.w 2
(B C).o
(A (B C) D).
```

2. The u (unwrap) command is the inverse of the wrap command. The current term must be a list, the effect is to remove the outer pair of brackets of the list. The first element of the list becomes the current term, and the other elements are inserted into the immediate context. To undo the effect of the wrap above we could perform the following sequence:

```
(A (B C) D).e
A.n
(B C).u
B.o
(A B C D).
```

Now we can see how to use these two commands to repair the various terms we showed above:

a) ((Prog A X) PR X Y)

This case is quite simple, we wrap up the sub-list PR X Y into a single list, so that it is put into the correct form:

```
[1]((Prog A X) PR X Y).e
(Prog A X).n
PR.w 3
(PR X Y).o
[1]((Prog A X)(PR X Y)).
```

b) ((PR A X)(PR) X Y)

The second example is a little more complex, a right bracket has been inserted too far to the left. To repair this we need to unwrap the list (PR), and re-wrap including the missing arguments:

```
[2]((Prog PR A X)(PR) X Y).e
(Prog A X).n
(PR).u
PR.w 3
(PR X Y).o
[2]((PR A X)(PR X Y)).
```



## 4. Utility modules

c) ((Prog A X(PR X Y)))

In this example we have first to wrap up the sub-list Prog A X to form an atom of the right form:

```
[3]((Prog A X (PR X Y)).e
(Prog A X (PR X Y)).e
Prog.w 3
(Prog A X).o
((Prog A X)(PR X Y)).
```

Now we have one too many pairs of brackets at this level, so we unwrap:

```
((Prog A X)(PR X Y)).u
(Prog A X).o
[3]((Prog A X)(PR X Y)).
```

This last unwrap has 'removed' the right bracket that was too far to the right.

The above three examples are used to illustrate the w and u commands. All three could also be handled using the t command at the top level to edit each clause as character text.

### 4.3.5 Further Extension

This editor represents a first attempt at the development of a term oriented structure editor for micro-PROLOG. Further possibilities for improvement are context searching and combining commands together with a repeat count. Since the editor is itself written in micro-PROLOG these enhancements should be quite straight forward.

## 4.4 Editing modules

We refer the reader to section 7.11 of Chapter 7 for a description of modules and the problem of editing a user program that has been wrapped up as a module. Only workspace programs can be easily edited using the above structure editor. So what is needed is a utility that allows modules to be unwrapped into workspace programs, and then, after editing, allows them to be wrapped up again as modules.

The file MODULES.LOG of the distribution disk contains such a utility program as the module modules-mod. To use it do a LOAD MODULES. To get rid of it, do a KILL modules-mod.

The module exports two unary relations wrap and unwrap that are used as commands. Both commands use the currently logged in disk for writing files. So this disk should have enough space for a file of all the clauses in the module and it should not be write protected. (The LOGIN primitive can be used to change the logged in disk and disk drive. See Chapter 7 or Chapter 3.)

The wrap and unwrap commands of this module are quite different from the w and u editor commands described above.

## 4. Utility modules

### 4.4.1 The unwrap command

To transfer the clauses of a module named M into the workspace for editing first SAVE any existing workspace program, then do a KILL ALL to clear the workspace, then do an

```
unwrap M
```

command. All the clauses owned by the module will enter the workspace along with an extra clause

```
((Module M <export list> <import list>))
```

which gives the name M of the unwrapped module and its export and import name lists. You can now edit or add to the clauses in the workspace using the structure editor described above. You can also edit the extra Module clause to change the name of the module or to change its export/import lists. But do not delete it. It is used by the wrap command to reconstruct the module.

The unwrap command uses the current logged in disk to store a temporary file which it deletes when all the clauses are in the workspace.

### 4.4.2 The wrap command

When you have finished editing the unwrapped module do a

```
wrap <file name>
```

command. This command uses the information in the Module clause in the workspace to create a module containing all the other clauses in the workspace. It also saves the new module on the named file.

By explicitly adding a Module clause to a workspace program you can use wrap as a quick way of creating new modules.

## 4.5 Using external relations - the EXREL utility

On occasions, especially when the definition of some of your relations comprises a lot of single atom clauses, there is very little space left when the entire program is loaded.

When space is at a premium, you can trade time for space and have some of your relation definitions reside in a specially created disk file. The file is created using the external command of the utility module exrel-mod which is in the program file EXREL.LOG of the distribution disk.

The external command saves all the current clauses for one or more relations in a disk file. It then deletes them from the workspace and adds a single new clause for each relation. This new clause 'gives' the segment of the disk file in which the defining clauses for the relation are to be found. During a query evaluation these defining clauses are automatically accessed from the file as and when required.

We now describe the commands and facilities of exrel-mod in more detail. It can be used to set up disk file data bases of micro-PROLOG relation definitions which interface with the query evaluation system in a transparent way.

To load the module do a Load EXREL command. To delete it do a Kill exrel-mod command. The module must be present whenever

#### 4. Utility modules

you want to query external relations. This is because it exports the definition of the interface program RPRED.

##### 4.5.1 external

This has two forms of use:

```
external <file name> <relation name>
```

sets up a file containing the defining clauses for a single relation.

```
external <file name> (<relation name1> .. <relation namek>)
```

sets up a file containing the current defining clauses for each of the list of relations. The disadvantage of packing several relations onto one file comes when we want to edit one of the relations - more on this later. The advantage is that it reduces the number of external files. Since micro-PROLOG only allows four files to be open at any one time, only four external files can be accessed during a single query evaluation.

The command deletes all the saved clauses from the workspace program. For each externally stored relation a single new defining clause is added. This links the relation with the segment of the file where the defining clauses were saved.

##### Example

Suppose that the workspace program contains a sequence of clauses of the form

```
((worker-ID <name list> <identifying number>))
```

e.g.

```
((worker-ID (Tom Jones) 2334))
((worker-ID (Bill J Bell) 2867))
.
((worker-ID (Jane Roberts) 900))
```

and a sequence of clauses of the form

```
((salary <identifying number> <dept> <annual salary>))
```

e.g.

```
((salary 2334 invoicing 12000))
.
((salary 3456 design 14500))
```

and a conditional clause

```
((salary x y 8000) (LESS x 1000))
```

expressing the company 'rule' that all trainee clerks have an ID number less than 1000 and a fixed salary of 8000. (This is not meant to be realistic. The general rule is there to emphasise that any form of clause can be used in the definition of a

#### 4. Utility modules

relation that is made external. You are not restricted to relations defined only by single atom clauses.)

We can make the definitions for both relations external with

```
external EXFILE1 (salary worker-ID)
```

A file named EXFILE1.LOG will be created on the currently logged in disk drive and all the current clauses for the two relations will be saved on the file. They are deleted from the workspace and replaced by clauses such as:

```
((salary|x) (RPRED EXFILE1 0|0) (3|72) (salary|x)))
((worker-ID x) (RPRED EXFILE1 (3|72) (7|1) (worker-ID|x)))
```

These are the clauses for salary and worker-ID that you will see if you now try to LIST these relations. To see the old defining clauses you have to use the LISTEX command described below.

If you now want to pose queries that will access these two relations do an open EXFILE1 command. This is more fully described below. Basically, it opens the external file for access by RPRED (using the micro-PROLOG primitive OPEN) and records the fact that the file is open by adding an opened clause for the relation to the workspace.

##### 4.5.2 The RPRED relation

RPRED is a relation which converts a query condition for an external relation into a search through the clauses saved on a file segment. The segment is identified by its first three arguments - the file name, the start position and the stop position. As the segment is searched, each clause found is matched against the atom given as the fourth and last argument. The RPRED definition of salary given above can be read:

```
To solve any atom of the form (salary|x) (the x represents
any list of arguments)
find a clause in the opened file EXREL1.LOG in the
segment between character position 0|0) and character
position (3|72) that matches the atom (salary|x)
```

In general there will be several clauses in the segment that match (salary|x) for the given x, so there will be several solutions to the query.

The character positions are l-pairs of numbers because this is what is used by the file I/O primitives of micro-PROLOG (see Chapter 7). A single integer identifying the character position is not used because this would limit files to 64K characters. Chapter 7 also includes a description of the definition of a simplified version of RPRED in terms of the file primitives of micro-PROLOG.

##### 4.5.3 The open command and the opened relation

The use is

```
open <file name>
```

It is equivalent to the primitive OPEN command of micro-PROLOG

with the added feature that it 'remembers' the opened files by adding an opened assertion to the workspace. You can therefore find out which are the currently opened files by LISTING the opened relation. You should always use open before posing a query that will access an external relation. Use it instead of the OPEN primitive because the opened assertion it generates provides useful information for the various listing commands described below.

If you do not open the external relation file before querying any of its relations you will get an error. If you are using the errtrap-mod error handler described in Chapter 6 you will be able to recover from the error by opening the file and continuing. Otherwise, the query will be aborted. You must open the file and repose the query.

micro-PROLOG only allows four files to be open at any one time. This limits the number of different external relations you can access during a single query. You increase the number by putting more external relations on each file. When you have finished accessing all the external relations on a file you should close the file.

#### 4.5.4 The close command

The use is

close <file name>

It is the opposite of open. It closes the file (using the primitive CLOSE) and deletes the opened assertion for the file from the workspace.

#### 4.5.5 The internal command

This is almost the opposite of external. The difference is that it handles one relation at a time and leaves other external relations on the same file unchanged.

The use is

internal <relation name>

Notice that the file name is not given - it is not needed. The command picks up the clause defining the named relation in terms of RPRED, deletes this clause, and loads all the clauses on the file segment identified in the RPRED condition of the deleted clause.

The RPRED definitions for any other relations on that file are left unchanged, they can still access the appropriate segments of the file. However, there is now an unaccessible segment - a hole in the file - from which the definition of the internalised relation has been read. All its defining clauses are now in the workspace and can be added to or edited before the relation is again made external on another file. Well, it must be another file if there are still accessible relations on the old file on which it was kept. This is the disadvantage of packing more than one relation onto a file.

#### 4.5.6 The LISTEX, LISTFILE commands

If you LIST a relation that has been made external you see its RPRED definition. To see its actual defining clauses you need to use LISTEX. Its use is

LISTEX <relation name> where the named relation is external (i.e. is defined by an RPRED clause in the workspace program)

e.g.

LISTEX salary

will produce listing of all the old clauses for the salary relation. This command automatically opens the file on which the clauses are kept and will close it again unless there is an opened assertion for the file.

Hence the importance of using open rather than the primitive OPEN to open external files for querying. If you use open, listing the clauses for the external relation will not result in the file being automatically closed after the listing. It will be left open for the queries.

To see all the saved clauses on a file use LISTFILE.

LISTFILE <file name>

will list all the clauses on the file. This can be used for any file on which a micro-PROLOG program has been saved, not just the files holding the external relations. So you can use it to list a normally SAVED program. Again, the command automatically opens and closes the file unless there is an opened assertion for the file left by the open command.

#### 4.5.7 The listex and listfile commands

These are alternatives to the LISTEX and LISTFILE commands which can only be used when the Simple module described in the next Chapter has been loaded. They require the Simple module to be present because they use one of its exported relations to map the clauses on the file back into SIMPLE sentence form. Otherwise, they have exactly the same effect as LISTEX and LISTFILE. listfile should only be used to list a file of clauses that have been entered and saved using Simple, similarly listex should only be used to list the clauses for external relations entered as SIMPLE sentences.

#### 4.5.8 Changing the definitions of external relations

If you need to edit or delete some clause about a relation that is external then you must use the internal command described above to bring all the defining clauses back into the main memory. When you have made the changes, make the relation external again. You must use a different file if there are still external relations on the old one.

If the change to the definition is just the adding of some new clauses, you can just enter these in the normal way, leaving the old clauses on the file. If you ADDCL them before the RPRED definition for the relation, the new clauses will be used before

the file clauses for the relation.

You can add them so that they are positioned after the RPRD clause, or in such a way that the RPRD definition is sandwiched between new clauses. Wherever the RPRD definition is positioned it will be used by micro-PROLOG as though it were replaced by all the clauses that it points to on the disk file.

When you eventually do an internal command for the relation, the RPRD definition is deleted, no matter where it is, and all the external clauses are added at the end of the remaining clauses. A new external command will now save all the defining clauses onto the specified file.

## 5. SIMPLE\_PROLOG

SIMPLE provides a quite elaborate extension to the facilities of the built-in supervisor. It provides commands to add, delete and text edit programs and supports several high level forms of query. The most important feature is that it allows clauses to be entered and queries to be posed using a much more user friendly syntax than the basic syntax described in Chapter 2. The SIMPLE syntax for clauses is much closer to the syntax of English. SIMPLE is therefore a very suitable system to use for beginning micro-PROLOG programmers. It is the system described in the Primer.

The form of clause accepted by SIMPLE is called a sentence because its syntax is so different. The program development commands of SIMPLE compile sentences into clauses and map the clauses back into sentences when they are displayed. For most uses, the programmer does not need to know anything about the syntax of clauses, only the syntax of SIMPLE sentences. However, you can always see the sentences in their compiled clause form by using the LIST supervisor command. In contrast, the List command of SIMPLE maps the clauses back into sentence form before displaying them.

To use SIMPLE you execute a LOAD SIMPLE command. (Note the warning at the beginning of the next chapter on MICRO. None of the MICRO modules should be present when you load in SIMPLE.) The easiest way to enter the SIMPLE system is to invoke micro-PROLOG with a load simple in the CP/M command line as in:

```
A>prolog load simple
Micro PROLOG r.vv S/N *****
(C) 1982 Logic Programming Associates Ltd.
99999 Bytes Free
&&
```

The file SIMPLE.LOG which will be loaded contains four modules called:

```
Simple          (about 8K)
errmess-mod     (about 1/2K)
exptran-mod     (about 2K)
told-mod        (about 1K)
```

Simple is the module which defines and exports all the program development commands that compile and de-compile SIMPLE sentences. Note that it does not have the usual "...-mod" form of module name. This is for compatibility with the Primer which uses the name Simple (see Chapter 7 of the Primer).

errmess-mod is a very simple error handling module. It exports the definition of a relation called "ERROR?". When this relation is defined the micro-PROLOG interpreter calls its program on encountering a runtime error. The ?ERROR? program in errmess-mod displays the condition currently being evaluated together with a more meaningful error message than the default error messages. For more information on error handling consult Appendix A. The messages given by errmess-mod are very similar to those of the example error handler given in that appendix.

The module can be killed and replaced by the `errtrap-mod` error handler described in 6.3 of the MICRO chapter, or by one called `deftrap-mod` which is described below.

`exptran-mod` is the module which compiles expressions if they are used in a sentence. The ability to use expressions in sentences is a feature of the SIMPLE for version 3 of micro-PROLOG. If expressions are not used it can be killed saving about 2K.

`told-mod` defines and exports the relation `is-told`. This can be used in a straightforward way to write interactive programs, programs that ask for information by posing queries to the user. The `is-told` program can also be invoked from the alternative error handlers `deftrap-mod` and `errtrap-mod`. Again the module can be killed if `is-told` is not used.

The SIMPLE system described in this chapter is the SIMPLE for version 3 of micro-PROLOG. It is an elaboration of the system described in the Primer but is completely compatible with it.

### 5.1 Syntax of sentences accepted by SIMPLE

A sentence is either

(i) a simple sentence

or (ii) a conditional sentence.

**Simple sentences** have two forms: binary and non-binary.

**Binary simple sentences** have the form:

`<term> R <term>`

where `R` is a constant which is a relation name. The terms on either side of `R` are the two arguments of the binary relation.

e.g. John likes Mary  
x LESS 45

**Non-binary simple sentences** have the form:

`R( <term> <term> .. <term> )`

where `R` is a constant which is a relation name. The list of terms following `R` are the arguments of the relation and are separated only by spaces. Sentences about binary relations can be entered in this non-binary form but will always be displayed by SIMPLE in the binary form.

e.g. SUM(X 5 34)  
APPEND( (2 3) (4 5) X)

**Conditional sentences** have the form:

`<simple sentence> if <compound condition>`

A **compound condition** is

(i) a condition

or (ii) a conjunction of the form:

`<condition>` and `<compound condition>`

`&` is an accepted abbreviation for and.

a condition is

(i) a simple sentence

or (ii) a **negated condition** of the form:

`Not( <compound condition> )`

or (iii) an **Is-All condition** of the form:

`<term> Is-All ( <term> <compound condition> )`

or (iv) a **For-All condition** of the form:

`<compound condition> For-All  
( <list of vars> <compound condition> )`

or (v) an **equality condition** of the form

`<expression> = <expression>`

or (vi) an **expression condition** of the form:

`R ##(<expression> <expression> ... <expression>)`

`is-all` and `for-all` are accepted synonyms of `is-all` and `for-all` when sentences are entered, but the conditions will always be displayed using `is-all` and `for-all`. When the `<list of vars>` of a `For-All` just contains one variable the list brackets are not necessary.

Expressions are terms of a special form. The syntax of expressions is given in 5.3 below.

The syntax of terms is as described in Chapter 2. There is one difference. The control characters that are allowed in quoted constants in clauses should not be used in sentences. If they are, they will not be displayed in the correct form by the SIMPLE List command and may well upset the way the sentence is displayed.

### Example conditional sentences

`PRED(x y z) if PQ(x y z) and y LESS z`

`x likes y if female(y) and Not(y likes Peter)`

`x GE y if Not(y LESS x)`

`PRED(x y z1) if x LESS y and Not(PR(y z x) & QUALIFY(x))  
and z1=(x#y+z)`

`x all-the-sons-of y if x Is-All (z y parent-of z & male(z) )`

`only-has-sons(x) if (male(y)) For-All (y x parent-of y)`

## 5.2 The relations and commands defined by Simple

We briefly describe the program development commands and other relations defined in and exported from Simple. For a more tutorial introduction we refer the reader to the Primer. All the commands have a completely lower case synonym. This is a feature only of the SIMPLE for version 3 of micro-PROLOG.

### 5.2.1 Add (or add)

The Add command allows you to add a sentence into the workspace. The sentence to be added must be enclosed in brackets. (Thus the argument to Add is a list of terms that conforms to the syntax of a sentence.) One form of the command is:

```
Add (<sentence>)
```

e.g. &. Add (Peter likes x if Not(x likes John))  
&.

The sentence will be compiled into a clause and added to the end of the current list of clauses for the relation that the sentence is about. (If the sentence is a simple sentence this is the relation of the sentence, otherwise it is the relation of the conclusion of the conditional sentence.)

To add into the middle of a program the form

```
Add n (<sentence>)
```

is used, where the number n refers to the position in the current sequence of clauses for the relation after which the new clause is to be added. For example, to add to the beginning of the likes relation use

```
&. Add 0 (Peter likes John)  
&.
```

Whenever you add a sentence about a new relation the relation name will be recorded in a dict assertion which is automatically added to your workspace program. So to see all these relation names you List the dict relation.

### 5.2.2 List (or list)

The List command displays the program on the console. To display the whole of your program type

```
&. List All  
Peter likes John  
Peter likes X if  
Not (X likes John)  
&.
```

list all is an accepted equivalent. Note that the clauses are displayed in sentence form. The List command de-compiles the clauses back into sentences before displaying them. You will see all the sentences for the relations recorded in the dict

relation.  
To display just a single relation, the likes relation say, use

```
&. List likes
```

This uses the alternative form of the command:

```
List <relation name>
```

Note that this means that you should not use all or All as relation names.

To print a program on the printer use the ^p (control-P) toggle before Listing the program. This will automatically print the program on the printer as it is displayed on the screen.

```
&.^pList All  
: {listing of the program copied on to the printer}  
&.^p
```

The final ^p turns off the copying of the screen display onto the printer.  
To see all the currently defined relation names use:

```
&.List dict
```

You will see all the relation names about which you have Added a sentence, providing you have not Killed the relation.

### 5.2.3 Delete (or delete)

This deletes a single clause from the program. It has two forms of use:

```
(i) Delete (<sentence> )  
(ii) Delete <relation name> n  
where n is the position of the sentence to be deleted.
```

e.g. &. Delete likes 3  
&. Delete (tom likes mary)

5.2.4 Kill (or kill)

Kill will delete an entire relation using the form:

```
Kill <relation name>
```

e.g. to delete all the clauses for the likes relation type

```
&. Kill likes
```

To get rid of the entire workspace program (you should normally only do this after a Save) use

```
&.Kill All (or kill all)
```

Finally, to get rid of a module use the form : Kill <module-name>  
 &.Kill expran-mod

Will get rid of the expression parser if you do not want to use expressions in sentences.

You will be queried about the Kill All use and asked to confirm that you want the entire program deleted. All the uses report the successful deletion of the program or module. When a program for a relation is Killed its entry in the dict relation is also deleted. After a Kill All the dict relation will be empty.

#### 5.2.5 Does (or does)

The Does command makes a YES/NO query of the program. It has the form:

Does ( <compound condition> )

with the single argument a bracketed compound condition. If the compound condition can be solved, the response is YES otherwise it is NO.

For example,

```
&. Does (John likes Mary)
YES
&. Does (SUM(2 3 x) & x LESS 5)
NO
```

#### 5.2.6 Which (or which)

The Which query finds all answers to some condition of a specified form. The syntax of the command is

Which ( <term> <compound condition> )

where <term> denotes the form of the answer required, and the <compound condition> is the query to be evaluated. For example, to find all the people that like John use:

```
&. Which(x x Likes John)
Answer is Peter
Answer is Mary
No (more) answers
&.
```

To find the pairs of people who like each other:

```
&. Which ((x y like each other) x likes y and y likes x)
Answer is (Peter Mary like each other)
No (more) answers
```

Notice the use of the constants in the answer term to make up an answer message. The answer term can be of any length or form.

To compute the sum of 3 and 5

```
&.Which(x SUM(3 5 x))
Answer is 8
No (more) answers
```

#### 5.2.7 One (or one)

The One query is similar to Which except that it prompts after each solution. If you respond with C or c then the next solution is sought, any other response stops the search for solutions:

```
&.One(x x likes John)
Answer is Peter.C
Answer is Mary.S
No (more) answers
&.
```

#### 5.2.8 Save (or save)

The entire workspace program can be saved for later use with this command. It is saved in clause form, not in sentence form. The use of the Save command is:

```
&. Save <file name>
```

where <file name> is a file name in the normal micro-PROLOG form. (See section 3.3.6.1)

The <file name> must be different from any relation name in the program. The Saved program can subsequently be reloaded using Load.

#### 5.2.9 Load (or load)

The Load command is used to re-load a previously Saved SIMPLE program. Its use is

```
&. Load <file name>
```

#### 5.2.10 Accept (or accept)

If entering a lot of data the Accept command can be used as an aid in adding a lot of simple sentences for a relation. It enables a sequence of simple sentences to be added without the need to repeatedly use the Add command. An example use is:

```
&. Accept likes
likes.(John Mary)
likes.(John Peter)
likes.End
&.
```

The Accept command prompts for a simple sentence (in non-binary form) with the name of the relation involved. The list of arguments of the sentence are then entered. You can continue entering sentences in this way until you enter End or end. The above example is equivalent to:

```
&.Add(likes(John Mary))
```

```
&.Add(Likes(John Peter))
```

#### 5.2.11 Edit (or edit)

The line editor can be used to edit an individual sentence by using this Edit command. (See Chapter 3 for details of the line editor) The Edit command is invoked as follows:

```
&.Edit Likes 1 {the relation followed by the sentence number}
```

The system responds with the sentence (surrounded by brackets) which can then be edited in a similar manner to the command in the structure editor described in Chapter 4. Like List, Edit maps clauses back into sentence form before displaying them. It then re-compiles them into clauses on exit from the edit.

If you change the name of the relation that the sentence is about you will be asked if you want to add the sentence to the program for the new relation name, and if so, at which position. You give the position you would give in an Add. If you do not want to add it to the program for this relation, you answer no and the edit will fail. That is, the old clause will remain in the program unchanged.

You can only Edit relations with names recorded in the dict relation.

#### 5.2.12 APPEND, ON, true-of

Simple includes and exports definitions of two useful list processing programs for the relation names APPEND and ON. In sentence form, their definitions are:

```
APPEND( ) X X
APPEND( (X|Y) Z (X|x))
  if APPEND(Y Z x)

ON(X (X|Y))
ON(X (Y|Z))
  if ON(X Z)
```

The APPEND relation has many uses. It can be used in exactly the same way as the appends-to relation of the Primer. The relation ON can be used to find members of a list or test for membership in the same way as the belongs-to relation of the Primer.

true-of can be used when the relation name of a condition is to be given as the value of a variable. Its form of use is:

```
<variable> true-of <list pattern>
```

The <variable> must be a relation name R by the time that the condition is evaluated. The <list pattern> represents the list of arguments for the relation. It gives the power of the predicate symbol meta-variable (see Chapter 2) to programs entered in sentence form.

For the clause form definition of true-of see the next chapter. The definition that is exported from Simple cannot be given in sentence form.

#### Example use

```
x true-of-all Y if (x true-of (y)) For-All (y y ON Y)
```

defines a relation true-of-all that can be used to test if all elements of a list Y have some property given as argument x. Notice that since x names a property - a unary relation - the second argument of true-of is a unit list (y) comprising the element to be tested.

```
Does( male true-of-all (Peter John Sam))
```

checks that Peter, John and Joan are all recorded as males.

```
Which(x dict(x) & x true-of (tom|y))
```

will find all the names of the relations recorded in dict such that tom 'satisfies' the relation. That is, if the relation is a property, then tom can be shown to have the property, and if it is a relation with more than one argument, then tom can be shown to be related to some other things by that relation.

The y in the list pattern (tom|y) represents the remaining arbitrary number of arguments of the relation x. When the true-of condition is solved y will be bound to a list of these extra arguments. When x is a property, y=().

The <list pattern> of a true-of condition can be a variable representing a list of any number of arguments. The use of variable in this way is not restricted to true-of conditions. Any simple sentence, written in non-binary form, can have its list of arguments represented by a variable. As an example,

```
Which(X father-of X)
```

can be used to find all the pairs in the father-of relation. It is a shorthand for

```
Which((x y) father-of(x y))
```

The use of a single variable as the argument following a relation is always a shorthand for a list of different variables, one for each argument of the relation. This, and true-of, are the only 'meta syntax' forms allowed in SIMPLE programs.

#### 5.2.13 CONS, @, &, function

CONS and @ are used in expressions which are dealt with in the next section. The & and = relations are only used together with expressions and are also described in the next section. Simple does not actually contain a definition of =, but it cannot be defined in a user program because it has a special role in the syntax of sentences. function is the command name that can be used to declare that a relation is a function so that it can be used in expressions. Again, its use and effect is described in the next section.



## 5. SIMPLE PROLOG

### 5.2.14 $\#$ , $\>$ , $\<$ , $\>$ , $\<$ PROD

The Simple module exports definitions for the auxiliary arithmetic relations  $\#$ ,  $\>$ ,  $\<$ ,  $\>$ ,  $\<$  and PROD. The auxiliary relations  $\#$ ,  $\>$ ,  $\<$ ,  $\>$ ,  $\<$  are recognised as arithmetic operators in expressions. The Simple definitions of the operators are:

```
+(X Y Z) if SUM(X Y Z)
-(X Y Z) if SUM(Y Z X)
*(X Y Z) if TIMES(X Y Z)
/(X Y Z) if TIMES(Y Z X)
```

PROD is the arithmetic relation for integer multiplication and division. It was a primitive of earlier versions of micro-PROLOG which only had integer arithmetic but it is not a primitive of version 3. It is therefore defined by a program in Simple that makes use of the new arithmetic primitives TIMES and INT. The PROD program is also given separately in the file PRODUCT.LOG.

The use of PROD is fully illustrated in the Primer. The new PROD is actually more general than as described in the Primer. Its three argument form is restricted to integers but its two four argument forms can be given floating point numbers to divide. The values returned are the whole number of times the divisor divides the dividend together with the remainder. Example,

```
Which((y z1) PROD(2.3 y 10.3 z1))
Answer is (4 1.1)
No (more) answers
```

## 5.3 Using expressions in sentences

Expressions can be used in equality conditions and expression conditions in sentences and queries. Such expressions are actually compiled into a relational form by SIMPLE; so that expressions are evaluated, during a query, by running the compiled form of the expression.

The expression compiler is contained in the module exptran-mod which is part of SIMPLE, and it is also contained in the file EXPTRAN.LOG. If you don't need expressions in your program you can save space by deleting the expression compiler.

### 5.3.1 Expression conditions

Expression conditions have the form

```
R # (E1 E2 .. Ek)
```

where R is the name of a relation and E1 ... Ek are expressions. Expressions are just terms of a certain form. The  $\#$  is the signal that E1 ... Ek are not normal arguments but that some or all of them contain arithmetic operators and function calls. The syntax of expressions is formally defined below. For now we shall just look at examples.

## 5. SIMPLE PROLOG

### Example

```
LESS #((2*x) (5+y))
```

is a LESS condition with arguments the values of the expressions (2\*x), (5+y). When it is evaluated, the variables x and y should have been given values. The relational form into which the condition is compiled is:

```
(X LESS Y) # (# (2 x X) and + (5 y Y))
```

The  $\#$  in this form should be read as where. So this condition is read as

```
X LESS Y where X is 2 * x and Y is 5 + y
```

This is the SIMPLE syntax form of what is produced by Expression-Parser. It is what will be displayed if you kill exptran-mod after the expression condition has been entered or if you add rel-form() to your program and list the sentence in which it is used (see 5.3.5 below).

The relational form of an expression condition

```
R # (E1 E2 .. Ek)
```

is

```
(R(t1 t2 .. tk)) # (<compound condition>)
```

The evaluation of the compound condition will produce values for the variables in the terms t1, t2, ... tk so that they become the values of the original expressions E1, E2, ... Ek. In our LESS example, the terms are the variables X, Y and the evaluation of the compound condition

```
# (2 x X) and + (5 y Y)
```

will result in X having the value of (2\*x) and Y the value of (5+y).

When a compiled  $\#$  condition is evaluated, the compound condition that produces the values of the expression arguments is evaluated first, then the R condition. On backtracking, only the R condition will be retried for alternative solutions, since there will be no alternative solutions for the evaluation of the expression values.

Example The condition

```
salary # (x (12*157))
```

is compiled into the relational form:

```
(x salary X) # (# (12 157 X))
```

The  $\#$  condition computes the value of (12\*157) and the evaluation of the  $\#$  condition will reduce to the evaluation of

```
x salary 1884
```

in order to find an x with recorded salary of 1884. Backtracking

will result in different values for  $x$  being sought but will not cause the re-computation of the value 1884. For details of the basic syntax form of a compiled expression condition and for the definition of # that is exported from Simple, and which is used to evaluate # conditions, see the next chapter. The MICRO definition of # relation is exactly the same as the Simple definition.

### 5.3.2 Equality conditions

An equality condition has the form

$E1 = E2$

where  $E1$  and  $E2$  are expressions. It is a shorthand notation for the expression condition

$EQ \# (E1 \ E2)$

Thus, the two expression arguments  $E1$ ,  $E2$  of an equality condition are evaluated and then their values are checked for identity using the primitive  $EQ$  relation.  $EQ$  is itself defined by the sentence

$EQ(X \ X)$

So, if one of the expressions is a variable, this will result in the variable being given the value of the other expression.

#### Examples

$x = (y \# 67 + z)$

can be used to give  $x$  the value of the bracketed expression if  $y$  and  $z$  have values at the time that the condition is evaluated. If they do not, there is a "control error" error message displaying either a call to  $SUM$  or a call to  $TIMES$ .

$0 = (2 * x * x + 7 * x - 23)$

can be used to check that the value of  $x$  satisfies the equation:

$2x^2 + 7x = 23$

(It cannot be used to find the roots of the equation)

### 5.3.3 Syntax of expressions

An expression is

- (i) an arithmetic expression
- (ii) a function call
- or (iii) some other term

An arithmetic expression is a list of the form  
 $\langle \langle \text{expression} \rangle \langle \text{operator} \rangle \langle \text{expression} \rangle \rangle$

where the operator is one of

# for multiply  
 % or / for divide  
 + for addition  
 - or ~ for subtraction (the ~ is safer because of the other syntactic roles of -. If you use - you should always surround it with spaces.)

The outermost brackets of an arithmetic expression are essential - so an arithmetic expression is just a three element list whose second element is an operator. However, if the expression arguments of this operator are also arithmetic expressions the inner brackets around them may be dropped in accordance with the following precedence

# / % equal - left associative  
 greater than  
 + ~ - equal - left associative

#### Examples

$(x \# y + 3 / z)$  equivalent to  $((x \# y) + (3 / z))$   
 $(x + y / (5 + z))$  equivalent to  $(x + (y / (5 + z)))$   
 $(x \# y / 5 + z)$  equivalent to  $((x \# y) / 5) + z)$

Simple exports definitions for the operators #, %, +, ~, -. Uses of / and ~ in expressions are mapped into conditions for # and - respectively. Any use of ~ or / in an expression will subsequently be displayed as a use of - or %.

A function call is a list of the form

$(R \ E1 \dots En-1)$

where  $E1 \dots En-1$  are expressions and  $R$  is a  $n$ -ary relation name that has been declared a function with the command

function  $R$

The expressions are the first  $n-1$  arguments of what would otherwise be given as an expression condition of the form

$R \# (E1 \dots En-1 \ x)$

The  $x$  found by the evaluation of this condition is the value denoted by the function call  $(R \ E1 \dots En-1)$ . Notice that this means that relations should be declared as functions only if the value of the last argument of the relation is uniquely determined by values for the preceding arguments.

**Example**

Suppose the relations `div` and `mod` are defined by the rules:

```
div(x y z) if TIMES(y z1 x) & INT(z1 z)
mod(x y z) if div(x y z1) & z=(zy#z1)
```

(Note the essential use of `~`. If `-` had been used the `z-y` would have been parsed as a constant resulting in the call to `TIMES` failing; this would cause the evaluation of `mod` to fail. However `z~y` is recognised by the lexical analyser as three terms `z`, `~` and `y` and hence as a use of the subtraction operator.

`INT` is a primitive of `micro-PROLOG` that can be used to test if a number is an integer or to find the integer part of a number, as here. So, `div(x y z)` can be used to find the integer divisor `z` of `x` and `y` and `mod(x y z)` can be used to find the remainder `z` of the integer division of `x` by `y`. For both `div` and `mod` the last argument is functionally determined by the first two arguments. So, we can declare these relations as functions

```
function div
function mod
```

and then use them in expressions:

```
x=((mod 85 23)*34)
LESS#((div 500 x) 23)
```

If you forget to declare that some relation `R` is a function before using it in an expression the expression parser of `exptan-mod` will assume that the function call is just a list that has as its first element a constant. However, it will tell you this by giving you the message

`R assumed not to be a function`

If the expression was in a query you will get the wrong answers. If the expression was used in an Added sentence you can easily recover from the error. Declare `R` as a function and Edit the sentence. Just call the line editor and then immediately exit it with `<return>`. The editor de-compiles the clause for the original sentence, mapping compiled expressions back into source form. It re-compiles the expressions on exit. This time it will recognise the use of `R` as a function call because of the declaration.

A function `R` declaration causes a `((func R))` clause to be added to the user workspace program. It is this that causes the expression parser to treat a list beginning with `R` as a function call when it appears in an expression. You can examine what functions have been declared with a

```
Which(x func(x))
```

query. Alternatively, you can list the `func` relation. If you Kill a relation that has been declared a function the `func` clause for the relation will be automatically deleted. (This only happens if you use the Kill or kill commands of Simple. If you use the supervisor Kill neither the `func` clause nor the dict clause for the relation will be deleted.)

### 5.3.4 Warning on the use of `!` in expressions

From the above discussion on the problem of undeclared function names it will be obvious that lists can be given as arguments to function calls in expressions. As an example

```
x = (APPEND (1 2) (APPEND (3 4) (5 6)))
```

can be used to append three lists. As we remarked earlier, the relation `APPEND` is defined in and exported from Simple. It is also recognised by the expression parser as a function name so you do not need to declare it as a function in order to use it in expressions. However,

```
x=(3 ! (APPEND y z)))
```

cannot be used to make `x` the list 3 followed by the concatenation of the lists `y` and `z`. This is because, as a `micro-PROLOG` term, the list

```
(3 ! (APPEND y z))
```

is just another way of writing the list

```
(3 APPEND y z)
```

in which the function call `sublist` has disappeared. So the expression parser sees a list of four elements and leaves it unchanged. The moral is that the term following a `!` in an expression can never be a function call. When you do want to denote the rest of a list by a function call you must use an explicit `CONS` function call (as in LISP) to construct the list instead of the primitive `!`. The above = condition needs to be re-expressed as

```
x = (CONS 3 (APPEND y z))
```

Like `APPEND`, `CONS` is defined in and exported from Simple and is recognised as a function in expressions. Its definition, in sentence form, is

```
CONS(X Y (X|Y))
```

Finally, there is one other predefined function. It is the apply function `@`. Its definition, which can only be given in clause form, is given in 6.1.16 of the next chapter. It applies its first argument, to the rest of its arguments. An example use is in the expression

```
x = (@ y 3 4)
```

If `y` is bound to `!` when this is evaluated `x` will be bound to 12. The `@` is needed. If you instead use

```
x = (y 3 4)
```

then `(y 3 4)` is not recognised as a function call. The equality is compiled into

```
EQ(x (y 3 4)) #()
```

with an empty condition for the argument evaluation. It will result in  $x$  being bound to the list ( $\#$  3 4) when evaluated. @ is the expression equivalent of the true-of sentence condition and the predicate meta-variable atom (see Chapter 2) for clauses.

### 5.3.5 When the expression handler is not needed

Expressions occurring in equality conditions and expression conditions are compiled into relational form using the relation Expression-Parse that is exported from the module exptran-mod. This same relation is used to de-compile the expressions when a clause is Listed or Edited.

If you kill the exptran-mod module after you have entered some sentences that used expressions the expressions will then be displayed in the compiled relational form even by the List and Edit commands of Simple. This is because a check is made to see if the module is present before Expression-Parse is used to de-compile expressions.

If the exptran-mod module is present then the expressions are displayed in the normal expression form. This means that you can have the convenience of using expressions in sentences whilst you are developing a program. Then, when you want to start using the program, you can get rid of exptran-mod with a

Kill exptran-mod

command to make available more space for the query evaluations. Of course, you will not be able to use expressions in the queries to the program.

If you do enter a sentence or query that uses expressions with the exptran-mod module not present you will get an error message of the form

```
No definition for relation
trying: Expression-Parse(<expression> <variable> <variable>)
```

If you are using one of the alternative error handlers deftrap-mod or errtrap-mod you can recover from this error and return to the interrupted compilation of the expression - more on this later. Otherwise you should load the exptran-mod module with a

Load EXPTRAN

command and re-enter the sentence or query.

Incidentally, you can see the compiled relational form of any expression in a program, even when exptran-mod is present, by adding the sentence rel-form() to your program with

```
&.Add ( rel-form() )
```

Now, when you List or Edit the program, compiled expressions will not be put back into expression form but will be displayed in relational form. (The relational form of an expression condition can be edited and it will be compiled back into clause form on exit from the Edit.) The display of the relational forms will continue until you Delete the rel-form sentence. It is useful if you want to check that you did not make a mistake in an expression and that what you intended is what has been recognised and compiled.

### 5.4 The error handler errmess-mod

On a runtime error the error handler errmess-mod that is loaded along with Simple just prints out a message identifying the error (the messages are similar to the ones given by the example error handler in Appendix A) along with the condition it was trying to evaluate when the error occurred. The current query or command is aborted and you are then returned immediately to the supervisor (you will get the &. prompt for a new query or command).

An alternative error handler, that allows you to recover from the very common error of not having a definition for a relation of the condition about to be evaluated (through forgetfulness or misspelling) is in the file DEFTRAP.LOG of the distribution disk. You can switch to this error handler by executing the following pair of commands, in the order given.

```
Kill errmess-mod
Load DEFTRAP
```

It is important that you do the Kill first. This is because errmess-mod and the module deftrap-mod that is loaded from the file DEFTRAP.LOG both export definitions for the error trap relation ?ERROR? (see Appendix A). If you do the Load first you will get the error: "illegal use of modules".

The only difference between errmess-mod and deftrap-mod is that the error "No sentences for relation" is specially handled by deftrap-mod. The condition that was being evaluated, which contains the undefined relation, is displayed followed by the prompt:

```
error&(? for info).
```

error& is the prompt you will continue to get whilst in this error state with the evaluation in which the error occurred suspended. Entering ? produces the message

```
to quit enter: q
or enter: tell (see manual)
or enter: / <any command> (eg / Add (sentence), / Load file)
to continue enter: c
```

If you enter any other first response than q / or c you will again get the ? info.

The q causes an ABORT to the top level supervisor which is what happens with the errmess-mod handler.

tell invokes is-told (see below) with argument the offending condition. It enables the current query evaluation. This is interactively during the development of programs. You can define useful for 'top-down' development of programs. You can define the higher level relations in terms of lower level relations whose programs are yet to be constructed. Then, when you query the partly developed program and get the "No sentences for relation" message for a lower level relation you can supply the answer to the condition using tell.

But note, your answers are not remembered. If a condition using the undefined relation is again encountered you will again get the error.

The only way to avoid this is to Add sentences defining the relation or to Load a file that contains a definition of the relation. You then enter the c to continue the suspended query evaluation. You can Add any number of sentences for the undefined relation before entering the c, in fact you can enter any number of commands providing each is preceded by the /. The / is the escape symbol that allows a command to be entered whilst in the error state.

Loading a file is the appropriate recovery response if you have Killed a module such as exptran-mod and you get an error message such as

```
No sentences for relation
trying: Expression-Parse((X*Y) Z x)
error&(? for info).
```

This will happen if you forgetfully use a = or & condition in a query, or in a sentence that you are adding to the program, after you have got rid of the exptran-mod module. The recovery response is

```
/ Load EXPTRAM
c
```

and the parsing will continue from the point that the error occurred.

A final example concerns the appropriate recovery action if you have misspelt a relation name, say you have used fatherof instead of father-of, you can recover by adding a sentence that defines fatherof as father-of. When the query evaluation is over you can edit the program to correct the spelling error and delete the definition of fatherof.

```
No sentences for relation
trying : fatherof (X bill)
error&(? for info). / Add (x fatherof y if x father-of y)
error&. c
```

Notice that the error handler displays the condition in non-binary form.

For an even more sophisticated way of handling errors, kill errmess-mod and load the MICRO error handler described in the next chapter. This is in the file ERRTRAP.LOG.

### 5.5 Using the relation is-told

is-told is a multi-argument relation exported from the module told-mod. When evaluated, an is-told condition displays its sequence of arguments followed by a ? and waits for a response. If there is just one argument, which is a list, this is displayed without the outer brackets. The responses and their effects are:

response effect

yes The is-told condition for the displayed argument is assumed to be true. Backtracking will not cause the question to be posed again.

The is-told condition is assumed false (i.e. it fails).

The .. is a sequence of terms, one for each different variable in the displayed message. The is-told condition is solved for values of the variables given in the response. The i'th term in the response sequence becomes the value for the i'th variable in the displayed message in the left to right order of the text.

Example, if the is-told condition is is-told(X likes Y) the message and response

```
X likes Y ? ans tom bill
```

makes X=tom and Y=bill. Backtracking will result in the message being re-displayed when an alternative solution can be given. This repeated prompting for new solutions on backtracking continues until you enter no or just.

just .. The same as ans except that on backtracking you are not asked for another solution. It is assumed to be the last solution to the is-told condition.

#### 5.5.1 Example use of is-told

(1) Which((percent z) is-told (mark x outof y) & z=(x/y\*100)) sets up an interaction that can be used to convert pairs of numbers to percentages. A example interaction is:

```
mark X outof Y ? ans 20 40
Answer is (percent 50)
mark X outof Y ? ans 15 60
Answer is (percent 25)
mark X outof Y ? just 40 120
Answer is (percent 3.3333333E1)
No (more) answers
```

(2) is-male (x) if known-male (x)  
is-male (x) if not(known-male(x)) & add ((known-male (x)))  
is-told (x a male) & add ((known-male (x)))

defines is-male in such a way that the user is queried whenever an is-male condition is encountered with argument given but not recorded as a known-male. A yes response to the question such as

```
keith a male ?
```

results in the is-male (keith) condition that provoked the question being solved and a known-male (keith) sentence being added to the program. A no response results in the condition failing.

## 5.5.2 Killing told-mod

If you do not want to use is-told in your programs or queries, nor use the tell response of the alternative error handler, you can Kill the told-mod module saving about 1K of program space. Even, if you do this and then forgetfully use the relation you can recover from the error by re-loading the module from the TOLD file if you are using one of the alternative error handlers described in 5.4. You can even recover if you use the tell error response with the told-mod module deleted. The interaction will be something like:

```
No sentences for relation
trying : father-of (tom X)
error&(? for info).tell
No sentences for relation
trying : is-told (father-of (tom X))
error&(? for info). / Load TOLD
error&c
father-of (tom X) ? .
```

and you can now continue by answering the father-of(tom X) question in the manner described above. You have recovered from an error encountered in an error recovery action! You are now out of the error state.

## 5.6 Using external relations - the EXREL utility

On occasions, especially when the definition of some of your relations comprises a lot of simple sentence facts, there is very little space left when the entire program is loaded. Incidentally, you can always find out the amount of free space left - for query evaluation or the loading of extra sentences - by querying the primitive SPACE relation.

```
Which(x SPACE(x))
```

will give you the amount of free space as a number of K bytes.

When space is at a premium, you can trade time for space and have some of your relation definitions reside in a specially created disk file. The file is created using the external command of the utility module exrel-mod which is in the program file EXREL.LOG of the distribution disk. This can be loaded and used in conjunction with the SIMPLE system.

The use of this utility is described in Chapter 4. Note that there are two commands supported by EXREL - listex and listfile that can only be used when the Simple module is present. They give a listing of an external relation and a program file in sentence form.

## 5.7 Tracing SIMPLE queries using SIMTRACE

The TRACE program described in Chapter 4 can be loaded and used to trace evaluations of programs developed using SIMPLE. The drawback of using this program, especially for beginning programmers, is that its ?? query form has the query expressed as a list of atoms and during the trace conditions are displayed as atoms. In fact, to use TRACE none of the SIMPLE modules need be present. They are best deleted to make available more space for

the trace.

There is a version of the trace facility called SIMTRACE, that makes use of the facilities of SIMPLE and allows the user to trace does and which queries posed using the SIMPLE syntax. It also gives more information during the evaluation - it gives information about the failed matches as well as the successful matches - and it displays the conditions being evaluated as SIMPLE syntax conditions. It is a useful program for a beginner to use to sharpen his understanding of how micro-PROLOG evaluates queries using backtracking. But because at least the Simple module must also be present, and because tracing takes up a lot of space, only relatively small programs can be traced using SIMTRACE.

The trace program is in the file SIMTRACE.LOG of the distribution disk. To use it do a Load SIMTRACE. It contains one program module called simtrace-mod. So to get rid of the tracer when you have finished do a Kill simtrace-mod.

The SPYTRACE program described in Chapter 4 can also be used to set up spyoints on programs developed and queried using SIMPLE. That there are spyoints on a program will not prevent the use of SIMTRACE. However, you will not be able to trace the evaluations of conditions for spyoint relations. You have to unspy the relation if you want to trace it using SIMTRACE.

## 5.7.1 The relations exported by simtrace-mod

simtrace-mod exports two relations: does-trace and which-trace. (Note they are both entirely lower case.) They are used in exactly the same way as the Does and Which queries of Simple. To trace the query

```
Which(x tom parent-of x & male(x))
```

use

```
which-trace(x tom parent-of x & male(x))
```

The trace will take you through the evaluation step by step. As each condition is reached, the condition is displayed in the form

```
<condition identifier> : <condition>
```

where the condition identifier is a list of integers that also gives the 'history' of the condition back to the original query. All the conditions of the original query have a single integer identification which is the position in the query. In the above query tom parent-of x will have the identifier (1) and male(x) the identifier (2). Whenever a rule is applied, the identifier grows by one number. The identifier (2 1) tells you that the condition is the second condition in the rule currently being used to evaluate the first condition of the original query. The identifier (3 2 1) tells you that it is the third condition of the rule currently being used to solve the second condition of the rule being used to solve the first condition of the original query.

When the condition displayed is for a relation in your program (one recorded in the dict relation) you will also get the prompt

```
trace ?
```

and the tracing will be suspended until you respond. The responses are the same as those for the TRACE of Chapter 4. They are:

y to trace the evaluation of the condition  
 n not to trace it, only the solutions to the condition will be shown  
 q to quit the trace entirely  
 s to resume the trace with the condition, as displayed, assumed solved  
 f to resume the trace with the condition, as displayed, assumed failed, i.e. assumed to have no solution

If you enter any other response you will be reminded of the allowed responses and prompted again.

The tracer will also prompt you when it reaches a complex condition such as an Is-All. This time the prompt will be

trace ?(y/n)

indicating that y and n are the only responses. In fact, any other response than y is taken as n. A y response enables you to trace inside the evaluation of the complex condition.

When a condition for one of your program relations is traced you will be told which sentence is being used to try to solve the condition with a message of the form

matching I : <condition> with head of N : <head of sentence>

where I is the condition identifier and N is the number of the sentence being used. (It is the position in the listing of the sentences for the relation of the condition.) You will be told whether the sentence matches the condition. If it does the result of the match will be displayed. Then, if the matched sentence has preconditions, these will be displayed as

new query : <precondition(s) of the matched sentence>

and the trace will continue with the evaluation of each of the conditions of the new query. The identifier for each of these conditions will be the identifier for the condition just matched with an extra condition number at the front.

When a condition is solved you will get the message

<condition identifier> solved : <condition in solved form>

When backtracking results in an alternative solution for a condition being sought, you will get the message

retrying <condition identifier> ..

Finally, when a condition cannot be solved, or when all solutions have been tried and the evaluation is backtracking to find alternative solutions to preceding conditions, you will get the message

<condition identifier> failing: <condition>

## 6. The MICRO extension to the supervisor

The MICRO.LOG file of the distribution disk contains four modules called micro-mod, errtrap-mod, exptran-mod and told-mod. The last three are also provided on the distribution disk in the single module files ERRTRAP.LOG, EXPTRAN.LOG and TOLD.LOG respectively. They can therefore be optionally killed and reloaded when needed. More about this later.

micro-mod is the main module. It exports several relations that can be used as auxiliary supervisor commands or directly in workspace programs when the module is present.

The commands are similar to those provided by the SIMPLE system of Chapter 5. The major difference is that the clauses that can be entered and edited using MICRO are essentially basic syntax clauses. The one elaboration is that expressions can be used as arguments to = and & conditions when exptran-mod is present.

As mentioned in the last Chapter, exptran-mod exports a relation called Expression-Parse that is used to compile expressions into relational form. This module can be deleted if the use of expressions is not required.

The errtrap-mod module should not be deleted unless it is replaced by an alternative error handling module. It exports the relation called ?ERROR?, which if defined, is always called by the micro-PROLOG interpreter when it encounters a runtime error (see Appendix A for more information). The error handling allowed by errtrap-mod is quite sophisticated and is described below.

Finally, the told-mod module exports a relation called is-told which can be used in a straightforward way to ask questions about certain conditions whilst a query is being evaluated. The use of is-told is documented in section 5.5 and so will only be briefly described in this chapter. It can be called from the error handler errtrap-mod in the same way that it can be invoked from the deftrap-mod error handler described in 5.4.

Any program developed under the SIMPLE system can be loaded and queried when using MICRO.

**WARNING** If you want to switch from using SIMPLE to using MICRO, or vice versa, you must first kill all the modules of the SIMPLE system before you load MICRO. This is because the modules of the two systems export common relation names. The attempt to load a module that exports a name already exported by another module gives an error. The safest thing to do when switching between the two program development systems is to save your user program, exit micro-PROLOG using QT., and restart using the other system.

### 6.1 The relations exported by micro-mod

To use the facilities of MICRO do a LOAD MICRO command. To get rid of it you must kill each of its modules with



## 6. The MICRO extension to the supervisor

KILL told-mod  
KILL exptran-mod  
KILL errtrap-mod  
KILL micro-mod

### 6.1.1 add

add has two uses:

- (1) add <clause>

will add the clause to the end of the list of clauses for its relation. Except when the exptran-mod module is present, and the clause contains expressions (see later), it is equivalent to just entering the clause in the manner described in Chapter 2.

- (2) add n <clause> n a positive integer

This adds a clause at the position indicated by the positive integer. It is equivalent to

?((ADDCL <clause> m)) where m is n-1

if the clause contains no expressions. This means that if there are already at least n-1 clauses for the relation the added clause becomes the new n'th clause. It is inserted before the old n'th clause if there is one. If there are not already n-1 clauses for the relation it is added as a new last clause. Notice that this is slightly different from the Add of SIMPLE.

For the MICRO add the number is the position before which the clause is added whereas for the SIMPLE Add the number is the position after which the clause is added. The difference is related to the different kind of clause editor in MICRO.

### 6.1.2 delete

This also has two usages:

- (1) delete <clause>

Again, except when the clause contains expressions, it is equivalent to

DELCL <clause>

with the added feature that if there is no such clause you get the message "No such clause".

- (2) delete <relation name> n n a positive integer

will delete the current n'th clause for <relation-name>. It is equivalent to

?((DELCL <relation name> n))

with the added feature that if there is no current n'th clause, it displays the message "No such clause".

## 6. The MICRO extension to the supervisor

### 6.1.3 edit

The edit command can be used to text editor a clause and/or reposition a clause. Its use is:

edit <relation name> n n a positive integer

The result is that the n'th clause for the named relation together with the number n will be displayed as

n <clause>

and both can be edited using the line editor described in Chapter 2. If n is changed, it is taken as the new position for the edited clause, i.e. the position that would be used in an add command. If the relation that the clause is about (i.e. the relation of the head atom) is changed then the new n is taken to be the position for adding the changed clause to the sequence of clauses for the new relation. The old clause is always deleted. The main difference between edit and the t command of the structure editor is that any compiled expressions in the clause are de-compiled back to expressions before the clause is displayed. This de-compiling only takes place if exptran-mod is present. If it has been killed then the clause will be displayed with all expressions in their compiled form.

### 6.1.4 cedit

Same form of use as edit. The difference is that the old clause is not deleted. Useful, for building up a definition of a relation where the clauses have common components. (c.f. the c command of the structure editor). As with edit, compiled expressions are de-compiled and displayed as expressions if exptran-mod is present. They are then re-compiled on exit from the edit.

### 6.1.5 kill

Same uses as the primitive KILL command of the supervisor. Only difference is that the use to delete entire workspace program is

kill all (note the lower case all)

and you are asked to confirm that the entire program is to be deleted. The program space that is available when it is cleared is also displayed. This space is the value given by the built-in micro-PROLOG SPACE relation. The other uses of kill are:

kill <relation name>  
kill <module name>  
kill <list of relation names>

They also report the successful deletion of the corresponding programs.

## 6. The MICRO extension to the supervisor

### 6.1.6 list

Again, same uses as supervisor command LIST with the exception that a request to list the entire workspace program is

list all (with lower case all)

Compiled expressions are displayed in their compiled form.

### 6.1.7 load <file name>

Same effect as supervisor LOAD <file name> command. The only difference is that after a load the free space left is given.

### 6.1.8 save <file name>

Exactly the same meaning as the supervisor SAVE <file name> command. It saves the entire workspace program on the named file.

### 6.1.9 reserved

A call (reserved x), x a variable, will result in x being bound to a list of all the names exported by the currently loaded modules. Useful for reminding you of the names of these exported relations which are names you cannot use for your own program relations.

The attempt to add a clause for a primitive relation or a relation exported by a loaded module results in the "Cannot add clauses for ..." error message. Nor can you avoid this error by adding clauses for the exported relation before you load the module. This will result in the "Illegal use of modules" error message. The programs inside modules are protected. To change them you should use the module utility described in Chapter 4.

### 6.1.10 space

The command

space. (the . or some other argument term is needed)

will give the current space left as a number of K bytes. It is equivalent to the query:

```
?((SPACE x))(PP x K free))
```

which uses the primitive SPACE relation.

### 6.1.11 ask

The use is

```
ask <list of atoms>
```

Except when the list of atoms contains expressions, which it compiles before evaluating the atom list, its effect is almost the same as the supervisor query command

```
? <list of atoms>
```

## 6. The MICRO extension to the supervisor

However, if the evaluation is successful, YES is displayed. If the evaluation fails, NO is displayed.

### 6.1.12 which. all

The use is

```
which (<term> <atom1> <atom2>...<atomk>)
```

Again, any expressions are compiled before the command is executed. The effect is the display of <term> for each different solution of the query

```
?(<atom1> <atom2>...<atomk>)
```

### Example

```
which((x y) (APPEND x y (1 2 3)))
```

produces the answer

```
((1) (1 2 3))
((1) (2 3))
((1) (2 3))
((1 2) (1))
((1 2 3) ( ))
No (more) answers
```

APPEND is a relation defined in and exported from micro-mod. See below.

all is an accepted synonym for which

### Example

```
all((y son of bill) (parent-of bill y) (male x))
```

produces an answer such as

```
(john son of bill)
(peter son of bill)
No (more) answers
```

### 6.1.13 one

The use is similar to which

```
one(<term> <atom1> <atom2>...<atomk>)
```

It will also display <term> for each solution of the sequence of atoms given in the query. The difference is that after each solution is displayed, it waits for a prompt to continue. If you enter c, it will give the next solution, if any. Any other response stops the evaluation. Again any expressions in the query condition are first compiled.

## 6. The MICRO extension to the supervisor

### 6.1.14 accept

accept enables a sequence of single atom clauses to be entered for a relation to be entered quite quickly. Its use is:

```
accept <relation name>
```

You will then receive the relation name as a prompt and you need only enter the list of arguments for the single atom clause that you want to enter. You can continue in this way until you enter end.

#### example

```
&accept male
male.(tom)      ((tom) is entered, male. is the prompt)
male.(bill)
male.(john)
male.end
```

will add the clauses

```
((male tom))
((male bill))
((male john))
```

to the end of the male program.

### 6.1.15 APPEND, ON, true-of

We have already mentioned that micro-mod defines and exports the list processing relation APPEND. Its definition is:

```
((APPEND () X X))
((APPEND (X|Y) Z (X|x))
 (APPEND Y Z x))
```

micro-mod also exports another useful list processing relation ON. It can be used to find members of a list or test for membership. Its definition in micro-mod is:

```
((ON X (X|Y)))
((ON X (Y|Z))
 (ON X Z))
```

true-of has the definition:

```
((true-of X Y)
 (X|Y))
```

You never need to use this relation in programs entered using MICRO. Any use of a call (true-of X Y) can be replaced by (X|Y). However, the relation may have been used in some sentence form of a clause entered using the SIMPLE extension of the supervisor described in the last chapter. It is included in micro-mod so that programs developed using SIMPLE can be loaded and queried when using MICRO. (See the remark at the beginning of the chapter.)

## 6. The MICRO extension to the supervisor

### 6.1.16 CONS, &, &, &, function

The definitions of the CONS and @ relations are:

```
((CONS X Y (X|Y)))
((@ X | Y)
 (X | Y))
```

They are used in expressions. The & and = relations are only used together with expressions and are described in the next section. micro-mod does not actually contain a definition of =, but it cannot be defined in a user program because it has a special role in the syntax of MICRO programs. function is the command that is used to declare a relation as a function before using it in expressions. (See the section 6.2.3 below and 5.3.3.)

### 6.1.17 &, &, &, = PROD

micro-mod exports the following definitions for the auxiliary arithmetic relations &, &, +, -, TIMES and SUM are primitive arithmetic relations of micro-PROLOG implemented in machine code. The auxiliary relations are recognised as arithmetic operators in expressions.

```
((+ X Y Z) (SUM X Y Z))
((- X Y Z) (SUM Y Z X))
((* X Y Z) (TIMES X Y Z))
((% X Y Z) (TIMES Y Z X))
```

The definitions are the same as those given in 5.2.14 which are exported from SIMPLE.

The PROD relation is exported from micro-mod so that programs developed using SIMPLE can be queried using MICRO. The use of the relation is fully described in the Primer. PROD was a primitive of earlier versions of micro-PROLOG but is now defined in terms of the new primitive TIMES. The PROD definition is also contained in the file PRODUCTI.LOG of the distribution disk.

### 6.1.18 dir

dir is a command that will produce the list of all the files on the currently logged in disk whose names match some CP/M 'ambiguous' file name. Its use is

```
dir <ambiguous file name>
```

Examples are:

```
dir #           {will display a list of all the .LOG files}
dir *.#         {will display a list of all files}
dir tudors      {will check if TUDORS.LOG is on file and echo
name if it is. If it is not, nothing is displayed}
```

Remember that you can change the logged in disk using the primitive LOGIN and you can erase and rename files using the primitives ERA and REN. These are all described in Chapter 7.

## 6.2 Expressions in MICRO clauses

You can use expressions as arguments to certain calls in the body of clauses that are entered using the add, edit or credit commands described above. The expressions can be used as arguments to calls that use the two relation names = and #.

The three commands just mentioned scan each clause before adding it to the workspace. If any call has the relation name =, or the relation name # followed by a first argument which is a constant, the expression arguments of the call are compiled into a list of relation calls which becomes an argument of a compiled form of the original = or # call. You see this compiled form when you list or LIST the clause.

However, when you edit the clause using edit or credit the compiled call is mapped back into its source form before it is displayed. The delete <clause> command also compiles = and # calls before it scans for the clause to delete.

Expression arguments to = and # calls are also allowed in ask, which and all queries. Expressions are compiled and decompiled using the relation Expression-Parser which is exported from the module exptan-mod.

## 6.2.1 The # relation

Source calls to #, i.e. calls that will be compiled, have the form:

```
(# R E1 E2..Ek)
```

where R is a relation name and E1,...,Ek are expressions. The expressions are the arguments of the relation call, which is really to the relation R. The # signals that these arguments to R are not just ordinary terms, but are expressions, terms that contain calls to functions and arithmetic operations. An # atom in a MICRO clause is the equivalent of an expression condition in a SIMPLE sentence. The atom (# R E1 ... Ek) would be the expression condition R\*(E1 ... Ek)

## Example

```
(# LESS (2 * 3) (5 + 7))
```

is a call to the relation LESS with arguments the values of the expressions (2 \* 3) (5 + 7). This source call will be compiled into the target call

```
(# (LESS X Y) ((# 2 3 X) (+ 5 7 Y)))
```

in which a list of atoms that will evaluate the expressions (2 \* 3) (5 + 7) appears as the second argument. The first argument is the call to the relation LESS with arguments the variables which have the values of (2 \* 3) and (5 + 7) when this list of atoms is evaluated.

This # call is logically equivalent to the three ordinary calls:

```
(# 2 3 X) (+ 5 7 Y) (LESS X Y)
```

The definition of #, which is exported from micro-mod, is

```
((# X Y)
  (? Y)/
  X)
```

Thus # evaluates the atom list Y before it evaluates the call X. Note the use of the backtracking control primitive /. This prevents backtracking on the atom list evaluation, i.e. on the evaluation of the arguments to the call X. However, it does not prevent backtracking on the evaluation of X.

## Example

```
(# salary x (2 * y))
```

will be compiled into

```
(# (salary x z) ((# 2 y z)))
```

If the value of y is known at the time the call is evaluated, the # call will compute z and the evaluation of the # call will reduce to the evaluation of

```
(salary x N), N some number
```

Backtracking will result in different values being sought for x, but not in the re-computation of N.

## 6.2.2 The = relation

Uses of = have the form

```
(= E1 E2)
```

where E1 and E2 are expressions. It is equivalent to

```
(# EQ E1 E2)
```

i.e. to a call of the primitive EQ relation with arguments the values of the expressions E1, E2. An = atom in a MICRO clause is the equivalent of an equality condition in a SIMPLE sentence.

## Example

```
(= (2 * x) (3 + y))
```

is equivalent to

```
(# EQ (2 * x) (3 + y))
```

and will be compiled into

```
(# (EQ X Y) ((# 2 x X) (+ 3 y Y)))
```

The EQ relation is a primitive of micro-PROLOG and is defined by the single clause

```
((EQ X X))
```

So, a = call evaluates its arguments and then unifies them. When the values are numbers, as here, this amounts to checking that

## 6. The MICRO extension to the supervisor

they are identical. When one of them is a variable, as in

```
(= x (234/23))
```

it will result in x being bound to the value of the expression (234/23) i.e. 1.0173943E1.

### 6.2.3 Syntax of expressions

We refer the reader to section 5.3.1 for the syntax of expressions as well as for information about how a relation R may be declared a function, using the

```
function R
```

command, and then used in expressions. This command adds a

```
((func R))
```

clause to the workspace program. The func clause is not automatically deleted when you kill the relation R. So you should delete it yourself if you do kill a relation that has been declared a function.

As with SIMPLE, you can recover from the error of not having declared a relation as a function before using it in an expression in a clause. You will get the warning message from exptran-mod

```
R assumed not to be a function
```

edit the clause and immediately exit from the editor. The expression is de-compiled before it is displayed and then re-compiled on exit. This time the use of R as a function name will be recognised.

### 6.2.4 Killing exptran-mod

The exptran-mod module occupies between 2 and 3K of program memory. After it has been used to compile expressions in the clauses of some entered program it can be killed. You will not of course be able to use expressions in any of the queries to the program. If you do, or you forgetfully add a clause containing expressions when exptran-mod is not present, you will get an error of the form

```
No clauses for (Expression-Parse <expression> <var> <var>)
```

This is because the add command has found some use of an expression and has tried to call the Expression-Parse relation exported by exptran-mod. The error message has been displayed by the errtrap-mod error handler. You can recover from the error by loading the EXPTRAN.LOG file which contains just exptran-mod. The details are in the next section.

You will not get this error message when you try to edit a clause with compiled expressions even though Expression-Parse is normally also called by the edit commands to de-compile expressions. This is because before Expression-Parse is called to de-compile expressions a check is made to see if the exptran-mod module is present.

If exptran-mod is not present, the expressions are displayed

## 6. The MICRO extension to the supervisor

in their compiled form. This means that you can use exptran-mod whilst you are developing a program and want to use the convenient shorthand provided by expressions. Then, for serious use of the program where space might be at a premium, you can kill the module. You can re-load the module as required.

### 6.3 The error handler errtrap-mod

The module errtrap-mod exports the relation ?ERROR?. As described in appendix A, if this relation is defined by some loaded module or workspace program, the micro-PROLOG interpreter will call the ?ERROR? program when it encounters a runtime error. The program for ?ERROR? given in errtrap-mod then displays a message of the form:

```
<short phrase describing the error> <atom of the call>  
error&(? for info).
```

The short phrase for the error is the same as the one given by the example ?ERROR? program given in appendix A. The error& is the prompt that you will continue to get whilst in an error trap state. The query evaluation that caused the error is suspended. There are now various options that allow recovery action to be taken and the suspended evaluation to be resumed.

You can obtain a brief description of these options by entering ?. You will then get displayed:

```
to quit enter : q  
to fail call enter : f  
to succeed call enter : s  
to line edit call and resume enter : e  
or enter / <any command> (eg / add <clause> , / load file)  
or enter : tell  
to resume enter : c  
error&.
```

**q response** This quits the suspended evaluation and returns you to the supervisor. After the q you will get the normal supervisor prompt &. and you can use any of the supervisor or MICRO supported commands in the normal way.

**f response** The suspended query evaluation is resumed but with the error invoking call assumed to have no solution, i.e. to have failed.

**s response** Same as for f except that the call is assumed to have been solved with its current arguments.

**e response** The offending call will be re-displayed with the line editor of micro-PROLOG in edit mode. You can line edit the call in any way but if you want to leave variables in the call you should use the variable names displayed in the call. On exit from the line editor, the evaluation will be resumed but with the offending call replaced by its edited form.

**Note** you have not edited the program. You have only changed the call for this one execution in order to avoid the error. Generally, you should also have edited the clause that lead to the error using the / <command>

## 6. The MICRO extension to the supervisor

response, or you should edit the program when the current query evaluation finishes.

**/ <command> response** The / is the escape character that enables you to enter any supervisor command or MICRO command. This must be the command name followed by its arguments. (You cannot just enter a clause. You must use add or ADDCL to do this.) You can thus edit your program, list it, load files, or add new clauses. An example of a / load file recovery response is given below.

**tell response** This calls the is-told relation exported by the module told-mod with argument the offending call which can then be answered interactively in the manner described in section 5.5.

It is useful for top-down programming. You can use relations in your program before defining them. Then, when you get the error "No clauses for" when a call on the relation is reached you can provide the answer or answers to the condition in response to the is-told prompts without having to give the definition.

But note that your answers are not saved. A subsequent use of the relation in the current query may force you to give the same answers. The only way to avoid this is to load a file with a program for the relation, or to add some clauses to define it. The use of tell is also in many cases an alternative to editing the call.

**c response** This should only be given after you have executed one or more commands that will ensure that the error will not occur when the evaluation is resumed. For example, you have added some clauses for a relation after getting the "No clauses for" error message and are now ready to continue with the suspended evaluation. The evaluation is resumed at the point where the offending call was tried, so the call is re-evaluated.

If you enter any other response you will get the ? message.

### 6.3.1 Example error recovery

(1) Suppose that you have killed the exptran-mod that exports the Expression-Parse relation and then you use an expression in a query or added clause. You will get an error message of the form:

```
No clauses for (Expression-Parse <expression> <var> <var>)
error&( ? for info).
```

The . is the prompt for you to enter a response. If the file EXPTRAN.LOG is on the disk of the currently logged in drive (you can change the disk if need be using the LOGIN supervisor command - see Chapter 7 which describes all the primitive relations) then the following will recover from the error.

```
error&( ? for info). / load EXPTRAN
error&.c
```

(2) Suppose that you have misspelled a relation name in some call of a program clause, using say parentof instead of parent-

## 6. The MICRO extension to the supervisor

of. You will get an error message of the form:

```
No clauses for (parentof .. ..)
error&( ? for info).
```

There are several ways to recover.

(a) You can use e to edit the call and change parentof to parent-of. But remember this change only applies to this call. If the clause with the misspelled relation is used again you will get the error again.

(b) You can add a clause defining parentof as parent-of and then resume with the responses:

```
/ add ((parentof X Y)(parent-of X Y))
c
```

This avoids the error on this and all subsequent uses of the incorrect clause. After the current query evaluation is over you can find the misspelling, edit it and then delete this added clause. This is perhaps the best recovery response.

(c) You can list the clauses in which you think the misspelling appears and then edit the offending clause. This removes the problem for subsequent uses of the clause but not for any currently active uses which includes the current error. To avoid a recurrence of the current error, either do (b), deleting the new clause when the current evaluation is finished, or edit the call, or enter tell to invoke is-told with argument the offending call.

### 6.4 Using the is-told relation

is-told is a multi-argument relation exported from the module told-mod. When evaluated it displays its sequence of arguments followed by a ? and waits for a response. If there is just one argument, which is a list, this is displayed without the outer brackets. The responses and their effects are fully described in 5.5.

#### 6.4.1 Example use of is-told

These are the same examples as given in 5.5 but using the query commands of MICRO and its clause syntax:

```
(1) all( (percent z) (is-told mark x outof y) (= z (x/y*100)))
```

sets up an interaction that can be used to convert pairs of numbers to percentages. A possible interaction is:

```
mark X outof Y ? ans 20 40
(percent 50)
mark X outof Y ? ans 15 60
(percent 25)
mark X outof Y ? just 40 120
(percent 3.333333E1)
No (more) answers
```

## 6. The MICRO extension to the supervisor

```
(2) ((is-male x) (known-male x))
      ((is-male x) (NOT known-male x)
        (is-told (x a male)) (ADDCL ((known-male x))))
```

defines is-male in such a way that the user is queried whenever an is-male condition is encountered with argument given but not recorded as a known-male. A yes response to the question such as

keith a male ?

results in the (is-male keith) condition that invoked the query being solved and a ((known-male keith)) assertion being added to the program. A no response results in the condition failing.

### 6.4.2 Killing told-mod

If you do not want to use is-told in your programs or queries, or use the tell error response, you can get rid of the told-mod module saving about 1K of program space. Even, if you do this and then forgetfully use the relation you can recover from the error by re-loading the module from the TOLD.LOG file. You can even recover if you use the tell error recovery response with the module deleted. The interaction will be something like:

```
No clauses for (father-of tom X)
error&(? for info).tell
No clauses for (is-told ((father-of tom X)))
error&(? for info). / load told
error&c
(father-of tom X) ? .
```

and you can now continue by answering the (father-of tom X) question in the manner as described above. You have recovered from an error encountered in an error recovery action! You are now out of the error state.

## 6.5 External relations

The EXREL utility described in 4.5 can be loaded and used in conjunction with MICRO. The only constraint is that the listex and listfile commands must not be used as they require SIMPLE to be present. Their clause form equivalents LISTEX and LISTFILE can be used instead.

## 6.6 Tracing and Structure Editing

Both the TRACE and SPYTRACE utilities described in Chapter 4 can be loaded and used in conjunction with MICRO. If you use TRACE it might be useful to kill the extra modules exptran-mod and told-mod before tracing in order to gain space. You can re-load them after the trace. Alternatively, you can kill all four MICRO modules, and re-load all four with a LOAD MICRO after the trace.

The structure editor in EDITOR.LOG can also be loaded and used with MICRO. It is described in Chapter 4.

## 7. Built-in Programs

A special feature of the built-in programs in micro-PROLOG is that they model as closely as possible program defined relations. For example, the SUM relation can be viewed as though it were defined by a set of facts about addition, and the TIMES relation as though it were defined by the various 'times tables'. This is because the built-in programs attempt to simulate the different patterns of use of the relation; and the SUM built-in program is able not only to add up numbers, but also to subtract them.

For reasons relating to efficient implementation micro-PROLOG compromises on the ideal of supporting every possible use and generally allows only some of the possible uses of its built-in programs. In particular, the assembler coded built-in programs only support the deterministic uses of the relations they represent.

However, in general, each built-in program has several uses. This helps to minimise the number of names the programmer has to know, and also helps to keep micro-PROLOG programs 'invertible' - able to support different patterns of use for the relations they define.

If a particular call to a built-in program is an illegal use (for example if SUM is called with two or more arguments as variables) then the system raises a "Control Error". An error of this kind usually occurs only if there are too many variables in the call.

The 60 or so built-in programs are divided into a number of functional groups: the arithmetic operations, string operations, input/output operations, type predicates, data base operations, logical operators, module construction facilities, program library operations and miscellaneous programs. We take each group in turn and describe the formats and semantics of each built-in program.

For those relations that are implemented entirely as micro-PROLOG programs embedded in the supervisor we also give their defining program. It is the program that will be displayed if you LIST the relation.

In introducing each relation we give the form of use, a comment (inside "{}" brackets) which gives a brief description of its meaning, and the restrictions, if any, on its use. The restrictions concern the arguments that must be known, or the arguments that must be variables, when a condition for the relation is evaluated. When the argument can be any term we indicate this by using a "t" or "f" etc. in that argument position in the form of use. When the argument must always be known at the time of evaluation, and must be a particular type of value, we give the value type in the form of use. For example, if a particular argument to a relation must be an integer at the time of evaluation the type <integer> will appear in that argument position in the form of use.

Remember that each primitive unary relation can also be used as a supervisor command - see Chapter 3.



## 7.1 Arithmetic Relations

The arithmetic relations SUM, TIMES, SIGN, INT and LESS cater for the normal operations on integer and floating point numbers of addition, subtraction, multiplication, division and comparison.

## 7.1.1 SUM

(SUM x y z) (x+y=z) at least two arguments must be given, given arguments must be numbers

SUM is true of three numbers when the first two add up to the third. The numbers involved can be integer, floating point or a mixture of the two types. The SUM primitive can be used to:

1. **Check a sum.** If all arguments are given then SUM succeeds only if the first two numbers add up to the third. For example, (SUM 20 30 50) is true, as is (SUM 1.5 0.3 1.8) and (SUM 23.6 - 0.6 23).
2. **Add two numbers together.** If the first two arguments are numbers and the third a variable then the call succeeds by instantiating the third argument to the sum of the first two. For example, (SUM 30 -2 x) binds x to 28, and (SUM 30 2.5 y) will instantiate y to 32.5.

3. **Subtract two numbers.** If the third argument is a number, and either the first or the second is also a number (with the remaining argument a variable) then the call succeeds by binding the variable in the call to the result of subtracting the first number (or second) from the third. For example, (SUM x 3 15) binds x to 12, as does (SUM 3 x 15).

If an addition or subtraction results in an overflow (or underflow), then micro-PROLOG signals an arithmetic overflow (underflow) error. This causes any resident **ERROR?** program to be invoked as for other kinds of signalled error (see Appendix A).

micro-PROLOG uses 8 digits of accuracy in its arithmetic calculations, and the exponent can range from -99 to 99. Truncation (towards zero) rather than rounding is performed if the number of significant digits in a calculation is more than 8.

## 7.1.2 TIMES

(TIMES x y z) (x \* y = z)

at least two arguments must be given, given arguments must be numbers

The TIMES relation can be used to multiply, divide or to check a multiplication.

1. **Check a product.** If TIMES is called with all three arguments numbers, then the product of the first two numbers is checked against the third number. If they are the same then the call succeeds, otherwise it fails. For example, (TIMES 3 4 12) is true, as is (TIMES 0.5 -3 -1.5)

2. **Multiplication.** If TIMES is called with just the first two arguments given, and the third argument a variable, the call succeeds by binding the variable to the product of the two numbers. The call (TIMES 3 -4 x) results in x being bound to -12.

3. **Division.** There are two forms of the TIMES program which can be used for division, both involve the division of the third argument of the TIMES call by either the first or second argument depending on which is known at the time. The remaining argument is instantiated to the quotient of the two given numbers.

(TIMES x 10 30), x is bound to 3  
(TIMES 10 x 25) x is bound to 2.5

The division of an integer by another integer may of course result in a floating point number. Similarly if two floating point numbers are divided then it is possible that the quotient is a integer. Any necessary conversions between the two kinds of numbers are performed automatically.

If a division by zero is attempted then an "Arithmetic overflow" error is signalled.

## 7.1.3 LESS

(LESS <number1> <number2>)

{<number1> is less than <number2>}

The LESS built-in predicate implements the inequality test for numbers. Only one arithmetic usage is allowed, where both arguments are given and are numbers. In this case the call succeeds if the first number is numerically less than the second; if they are equal or if the first number is greater than the second the call fails. For example, (LESS 2 3) is true, as is (LESS -1 1.32); but neither (LESS 10 9) nor (LESS 4.4 4.4) are true.

LESS can also be used to compare constants, see below.

## 7.1.4 INT

(INT <number> y) (y the nearest integer to <number>  
between 0 and <number>)

y must be a variable at call

The INT primitive can be used to find the nearest integer to a given number. The nearest integer (truncating towards zero) is returned as the value of y. This two argument form of INT can only be used in this single mode. A one argument form of use is described in 7.7.

The nearest integer to a large floating point number may not be representable as an integer in the range -32767 .. 32767. In this case a floating point number is still used to represent the result though it will have an integer value.

The nearest integer to 1.9 (towards zero) is 1: (INT 1.9 x) binds x to 1. The call (INT -134513.456 y) instantiates y to the number -134513, which because it cannot fit in 16 bits, is given as the floating point number 1.34513E5.

## 7.1.5 SIGN

```
(SIGN <number> y)      (<number> > 0,   y=1
                        <number> = 0,   y=0
                        <number> < 0,   y=-1)
```

y must be a variable at time of call

The SIGN primitive returns the sign of its numeric first argument in the second argument. Only one use is permitted where the first argument is a given number (can be integer or floating point), and the second argument is an unbound variable.

For example, (SIGN -3 x) instantiates x to -1, (SIGN 0 y) binds y to 0, and (SIGN 25.67 z) binds z to 1.

## 7.2 String operations

In micro-PROLOG strings can be represented and manipulated as lists of characters. A string in packed form is a constant. There is a primitive STRINGOF relation that can be used to convert between strings as character lists and constants. LESS can be used to test the lexicographical order of constants and CHAROF can be used to convert between characters and their ASCII codes.

## 7.2.1 LESS

```
(LESS <constant1> <constant2>)
```

```
{<constant1> lexicographically less than <constant2>}
```

Similar to the inequality test for numbers, this test for constants tests that the first argument (which must be a given constant) is textually less than the second (which must also be a given constant). The ordering used is the lexicographical ordering, based on the ordering of the underlying character set (namely ASCII).

For example, (LESS FRED FREDDY) succeeds since FRED is lexically less than FREDDY.

## 7.2.2 STRINGOF

```
(STRINGOF x y) (x is a list of characters of constant y)
```

x must be a list or list pattern and y a constant  
or x must be a character list and y a variable

The STRINGOF relation can be used to convert a constant into a list of its constituent characters, or to pack a list of characters into a single constant. There are essentially two forms of use:

1. **Unpacking.** to produce a list of characters from a constant. In this use the second argument must be a constant (not a number or list) at the time of evaluation. The result is the unification of the first argument with the list of characters of the constant. If the empty constant "" is given its list of characters is the empty list ().  
(STRINGOF x fred) results in x being bound to the list

(f r e d), and (STRINGOF x "A") binds x to the list (A #). For the unpacking use, the first argument may be given as a list or a list pattern. This allows comparison of a list of characters and the constant, and the use of a pattern allows a particular character of the constant to be picked up as the binding of a variable.

```
(STRINGOF (f r x d) fred), x = e
(STRINGOF (f rix) fred), x = (e d)
(STRINGOF (f rix) gerry) fails
```

A given list of characters must be just that: a list of constants which have single character names.

2. **Packing.** This takes a list of characters and produces a constant from it. It is the inverse of the unpack use.

```
(STRINGOF (f r e d) x) x = fred
(STRINGOF () x) x = ""
```

## 7.2.3 CHAROF

```
(CHAROF x y) (x is character with code y)
at least one argument must be given
```

The CHAROF primitive implements a mapping between single character constants and the ASCII coding sequence. Three uses are allowed: where either x or y or both are known at the time of the call. Some examples of its use are:

```
(CHAROF A 65) x is instantiated to 66
(CHAROF B x) y is instantiated to " "
(CHAROF y 32)
```

In fact, if the x argument is given it can be any constant. The ASCII code is then the code of the first character of the constant.

```
(CHAROF FRED z) z is bound to 70
```

Restriction CHAROF only handles ASCII codes 1 to 126. The two numbers 0 and 127 are not allowed values for the ASCII code argument.

## 7.3 Console Input/Output

The input/output facilities are divided into two groups: Console I/O and Disk I/O. Console I/O reads terms from the keyboard and displays them on the console. Disk I/O transfers terms to and from the disk system. In fact the console I/O primitives are defined in terms of the disk I/O primitives as we shall see below.

The I/O facilities described here are the first example of a non-logical feature of micro-PROLOG, this is because they depend on their behaviour (reading and writing terms) for their meaning. However, in large measure the power of PROLOG as a systems programming language arises from its combination of declarative and imperative features.

There are four built-in programs for dealing with Console I/O: R(Read), P(Print), PP(P-Print) and RFill, which respectively read a term from the console, print a sequence of terms, pretty print a sequence of terms and 'pre-fill' the keyboard buffer with a sequence of terms.

## 7.3.1 R

```
(R x)      (x the next term typed on keyboard)
```

x must be a variable at time of call

The Read program reads a single term from the keyboard and binds its argument to the term it reads in. It must be called with a variable as its argument, otherwise a "Control Error" is signalled. If there is already a term in the keyboard buffer (see Chapter 3) this will be the value returned, otherwise the read prompt . is displayed and the evaluation will suspend until a term is entered.

Any variables in the entered term are converted to the special internal form for variables in which the variable name used in the term is discarded. However, different occurrences of the same variable in the term will be converted into the same internal form.

The internal form for each variable of the term will be different from the internal form of any other variable in the program - i.e. it will be an entirely new variable. It will also be different from the internal form assigned to a variable of any other term that has been read in, or will be read in, even if the variables have the same name. In micro-PROLOG the scope of a variable name is the term in which it appears.

See Section 3.1 for details of how micro-PROLOG accepts terms from the keyboard.

The R primitive side-effects the keyboard buffer removing the first term from the buffer. A call to R only has one solution. A backtracking return to the call will not result in an attempt to find an alternative solution by reading in the next term. The second attempt to solve the call will fail. The call will be re-solved only if an earlier call has an alternative solution and this leads to a fresh attempt to solve the R call. At that point, the next term in the buffer will be read.

## 7.3.2 P

```
(P t1 t2 .. tk)  (display t1 t2 .. tk on the console)
```

The Print program displays its sequence of any number of arguments on the console output device. It takes any number of arguments each of which can be any term. The terms will be displayed separated by a single space. There is no automatic new line on completion. A subsequent P or PP will therefore start one character position after the preceding P finished its display.

The P primitive displays constants in its argument terms as the sequence of their component characters. Thus a constant that would have to be quoted on input (see Chapter 2) will be displayed without the quotes. This means that any control characters in the constant, which appear as @<char> combinations in the quoted form, are sent to the output device as control characters. A control character in a constant will therefore

produce the effect that it is supposed to produce on the display. For example, if the console uses Control-L to clear the screen then

```
(P "@L")
```

clears it. Using this unary form as a command, the command

```
P "@L"
```

will therefore also clear the screen. By putting other control characters in a quoted constant argument to P you can move the cursor to any position on the screen. What control characters are used depends on the terminal.

Variables occurring in the sequence of displayed argument terms are given the names X, Y, Z, x, y, z, X1,...,Z1,X2 and so on, corresponding to the order that the variables are encountered during the Print. (The first variable encountered is given the name X, the second one the name Y and so on.) The same internal form variable appearing in more than one term of the displayed sequence of terms will have the same print name. So, if the first variable that appeared in t1 also appears in t2, it will be displayed as X in both terms. If there are more than 128 different variables, then subsequent variables are displayed as ???.

## 7.3.3 PP

```
(PP t1 t2 .. tk)  (Pretty Print t1 t2 .. tk on the console)
```

The PP program displays its sequence t1 .. tk of arguments in a standard format. It is very similar to the P program except that a carriage return/line feed automatically occurs after the terms are displayed. In other words, PP always terminates the display of its arguments by sending the <CR> and <LF> pair of characters to the terminal. A subsequent P or PP starts the beginning of a new line.

PP also displays in quoted form constants that would need to be quoted on input. This means that PP displays the terms in such a way as to guarantee that if they were read back in the same terms would be re-constructed (except for variables which are always new).

Control characters occurring in the quoted constants are displayed in the @<char> format. As an example of the difference between P and PP,

```
(P "(The man")
```

displays:

```
(The man
```

on the console, whereas the call:

```
(PP "(The man")
```

displays:

```
"(The man" <CR><LF>
```

## 7. Built-in Programs

on the console.

Since the PP program always finishes by sending a return/line feed pair a call to PP with no arguments:

```
(PP)
```

will just move the cursor to the beginning of a new line.

### 7.3.4 RFILL

```
(RFILL (t1 t2 .. tk) x)
```

{clear then pre-fill keyboard buffer with t1 t2 .. tk  
and then read a term into x}

x must be a variable at time of call

The RFILL program is used to invoke the built-in line editor with the sequence of terms t1 t2 .. tk given as the elements of its first (list) argument already in the keyboard buffer. The terms, as displayed, can then be edited using the line editor.

When <return> is entered the first term in the buffer (usually the edited form of t1) is automatically read in to become the binding of the variable x.

The edited forms of t2 .. tk, if they were given in the call, must be explicitly read in using R since they will be left in the keyboard buffer.

RFILL displays constants in the PP format so that they can be read back in. RFILL remembers the print names given to each internal form variable in t1. When the edited version of the displayed t1 is read back in as the binding of x, each of these remembered variable names is converted back into the internal form variable that appeared in the original t1. Thus, variables names in the edited form of t1, which are the same after editing, become the **same internal variables** as those that appeared in the t1 argument to RFILL.

This remembering of variable names is necessary for the use of RFILL for editing programs, particularly when it is used from within a structure editor to edit a component of a clause. RFILL is used in this way by the structure editor described in Chapter 4. It is invoked by the t command. RFILL is also used by the error recovery command of the errtrap-mod module described in Chapter 6. Again for this use, the remembering of variable names is essential.

Note that although a list of terms must be given as the second argument to RFILL, the list is displayed without the outer level of parentheses.

RFILL also has the side effect of clearing any previous contents of the keyboard buffer. If several terms had previously been typed on a line, then any 'unused' terms will be discarded when RFILL is invoked.

As we have already mentioned, RFILL is used by the t command of the structure editor. It is also used by the edit commands of both SIMPLE and MICRO. As an example, a simplified version of the edit of MICRO is defined by:

```
((edit x)
 (R y)
 (CL ((x|z)|z1) y y)
 (RFILL ((x|z)|z1) z)
 {command: edit relation x}
 {read the clause number}
 {find the y'th clause of x}
 {invoke line editor with the
```

## 7. Built-in Programs

```
clause in the buffer)
(ADDCL z y)      {add the edited clause z to
                  the program}
(DELCCL x y))     {delete the old clause for x}
```

For an explanation of the other primitives see Section 7.9.

### 7.4 File I/O

micro-PROLOG supports files of text under CP/M, i.e. files of ASCII characters. These are accessed sequentially via the primitives READ, W and WRITE or randomly using the SEEK primitive. Up to four files may be open during an evaluation at any one time, an attempt to have more files open results in the "Too many files opened" error being signalled.

#### 7.4.1 OPEN

```
(OPEN <file-name>) {open named file for reading}
```

This built-in program opens a file for reading. The <file-name> argument is a constant which names a file according to the CP/M file naming conventions (see Chapter 3 for details).

If the file was previously open for writing then the file buffer is 'flushed' and the file is closed before being re-opened for the read. This means that it is not possible to simultaneously read and write to a file. The first character of the file is read in by the OPEN program, so if the file is empty or if it is not there then the "File not found" error is signalled and the OPEN call fails.

If the file was already open for reading then the file is 'rewound' to the beginning; however it is not good practice to rely on this feature.

#### 7.4.2 CREATE

```
(CREATE <file-name>)
```

{create and open a new file for writing}

The CREATE program opens a new file for writing. Any old file of the same name is first re-named to have extension .BAK, and the new file is then created. This means that files are automatically backed up.

If the file is already open for writing then the "File error" error is signalled.

#### 7.4.3 CLOSE

```
(CLOSE <file-name>) {close down the named file}
```

In CP/M there are no special actions associated with closing a file which is being read; however a file which is being written to must be explicitly closed down, otherwise the disk file may not contain the right data.

The CLOSE primitive performs this operation and releases the file from micro-PROLOG. It is also used to release, from micro-PROLOG, files opened for read access.

## 7.4.4 READ

```
(READ <file-name> x) {x next term on the named file}
```

x must be a variable at time of call

The READ primitive reads the next term from the named file and binds its argument to the term. It finds the term by reading characters from the current character position for the file. If the current character position is at the end of file, the READ fails. After the read, the current character position is immediately after the last character of the read-in term. The READ primitive is the same as the R primitive, except that it reads a term from a named file.

## 7.4.5 WRITE

```
(WRITE <file-name> (t1 t2 .. tk))
```

```
{write the sequence of terms t1 t2 .. tk on the named file
 in PP format}
```

The sequence of terms t1 t2 .. tk are written onto the file in the same form as the PP program. This ensures that any term written onto a disk file can be subsequently read back in as the same term, with the exception of variables which are renamed. At the end of the transfer of t1 .. tk a line feed/return two character sequence is sent to the file. So the file position at the end of the WRITE is not immediately after the character text for the last term tk, but is two more characters beyond that.

## 7.4.6 W

```
(W <file-name> (t1 t2 .. tk))
```

```
{write the sequence of terms t1 t2 .. tk on the named file
 in P format}
```

Exactly the same effect as WRITE except that the terms are written in the same way that P displays terms on the console. Note that terms written using W will not be re-readable as terms if they contain any constants that would need to be quoted on input. The W does not complete the transfer with a line feed/return. The file position on completion is therefore immediately after the text for tk.

Both W and WRITE handle variables in the same way as the P and PP primitives. Each occurrence of a variable throughout the list of terms (t1 .. tk) is given the same print name in the sequence of terms written to the file.

## 7.4.7 INTOK

```
(INTOK <file-name> x) {x is next token on named file}
```

x must be a variable at time of call

The INTOK primitive reads the next token from the named file and returns it as the value of its second argument, which must be a variable. The lexical rules used for defining tokens in character files are the same as those for micro-PROLOG terms; for

a complete description of the lexical rules see Section 2.10. If the read-in token represents a number, its number value will be the binding of x. For example, if the next token is the single character 9, then the integer 9 will be the value of x. For any other token, the value of x becomes a constant comprising the characters of the token. Even if the next token read in would normally be interpreted as a variable, a constant comprising the characters of the variable name is the value of x. If such a constant is pped, then it will be surrounded with quotation marks to signify that it is actually a constant and not the print name of a variable.

INTOK is therefore useful when it is important that the normal micro-PROLOG variable conventions should be ignored; in particular if a y response is read in using a call

```
(INTOK "CON:" x)
```

as an allowed shorthand for yes, then the constant y is the value of the variable x and a subsequent (EQ x n) test will fail. If (R x) is used, the y response is interpreted as a variable, and x effectively remains unbound. A subsequent (EQ x n) test will now succeed, binding x to n.

For an explanation of the use of the file name CON:, see 7.4.9 below.

## 7.4.8 SEEK

```
(SEEK <file-name> x) {named file is at position x}
```

x a variable or file position at call

There are two modes of use, either the x argument is unbound on entry to SEEK, or it is a file position previously returned by an earlier call to SEEK.

A file position is a 1-pair of integers (x,y). The x is a 'block' number within the file. Each block is 128 characters long and a file can contain up to 32K blocks. The y is an integer 'offset' (starting at zero) into this block. The offset is a character position within the block. Hence the first character on a file is in the zeroth block, and has offset zero; it is at position (0,0).

If the position argument is unbound then SEEK returns the current position in the file; if the argument is given then the file is re-positioned to the position given.

The SEEK primitive can be used for files that are either opened for reading or for writing. However, extreme care should be exercised if writing into the middle of a file, as there is no protection against overwriting already existing terms on the file. SEEK can be used to implement a system where one or more

programs can be on disk instead of in memory. The following is a simplified form of the RPRED definition exported by the EXREL utility described in Chapter 5.

```
((RPRED x y z)
  (find clause with head y on opened
   file x starting search at position
   z)
  (move file to z)
  (read in a candidate clause y1)
  (find new file position z1)
  (either)
  ((EQ (y1Y) y1)Y)
  (unify candidate clause y1 with
   (y1Y) and evaluate the body of the
   clause Y)
  ((RPRED x y z1))) (or find another clause starting at
  z1)
```

To use this program (ignore the comments between the curly brackets) to keep the clauses for a relation likes on the disk file LFILE.LOG replace the clauses for likes in the program with the single clause:

```
((likes|x)
  (RPRED LFILE (likes|x) (O|O)))
```

The file LFILE must also have been OPENED prior to using the program for likes. The LFILE file can be generated either by using SAVE (see below) or be specially generated by a program that uses WRITE.

The likes clauses in LFILE can be quite general, including recursive clauses, because different invocations of RPRED can be accessing different segments of the file simultaneously. The use of SEEK means that each different use of RPRED remembers where it is in the file, and then explicitly repositions the file to that position when it is getting its next clause.

You can spread the clauses for a relation over several files (just use more than one rule as above) and you can mix RPRED definitions of relations with ordinary clauses. So, new information about a 'data base' relation can be recorded by a clause in memory until the backing store file can be updated. Programs like RPRED can be used to build intelligent data base systems within micro-PROLOG.

#### 7.4.9 Special file names

The four 'serial' input/output channels under CP/M are denoted by special file names. These are:

```
"CON:" the console device (I/O)
"LST:" the printer device
"PUN:" the punch output
"RDR:" the reader input.
```

These special file names can be used in any of the above file I/O primitives, except for SEEK which demands a disk file. Thus if a given CP/M installation connects the LST: device to a parallel printer then WRITING to the LST: file will actually print directly on to the printer.

Similarly the RDR: and PUN: devices are commonly implemented as serial input/output communication (RS232). It is quite possible to have two computers both running micro-PROLOG programs communicating with each other by sending messages via the RDR: and PUN: devices.

If the RDR: device is used then it is important to ensure

that the serial device is interrupt driven. That is the characters coming in from the serial device are queued, and whenever a new character appears on the serial channel the computer is interrupted and the new data automatically put into the queue. **This is not done by micro-PROLOG.**

The automatic queueing is necessary because micro-PROLOG cannot always guarantee to keep up with the incoming information. The the maximum baud rate which can be used is 1200baud on a 4MHz Z80. If a higher baud rate is used, or if the garbage collector happens to be called during the read, characters will be lost and ignored.

Finally, the R, P and PP commands are defined by the following supervisor programs using READ, W and WRITE and the special file name CON:.

```
((R X)
  (READ "CON:" X))

((P|X)
  (W "CON:" X))

((P|P|X)
  (WRITE "CON:" X))
```

Note how the use of the meta-variable as the list of arguments (see Chapter 2) makes P and PP multi-argument relations. The sequence of argument terms given in the call is passed on as a single list of terms to the W and WRITE primitives.

#### 7.5 File operations

A number of primitives allow the manipulation of CP/M files directly; such operations include renaming files, erasing them and logging in a new disk.

##### 7.5.1 LOGIN

```
((LOGIN <disk-name>))
```

{make disk in named disk drive the logged in disk}

CP/M has an annoying habit of not allowing you to easily change disks. If you do change a disk without telling the system in some way (you have to type control-C on CP/M's top level prompt), then the new disk is marked "read-only" and you cannot write on it. If you attempt to write on the disk (for example by trying to save a PROLOG program) CP/M aborts the micro-PROLOG system and returns you to its top level prompt.

What the LOGIN primitive allows you to do is to change a disk without leaving micro-PROLOG.

The <disk-name> is a single letter constant which is one of the CP/M disk drives. Examples of disk names are A, B .. P. The LOGIN primitive logs in the disk named by the parameter, and makes it the new default drive. The default drive is where micro-PROLOG (and every other CP/M program) looks for a file if the disk drive is not explicitly given in the file name.

As a supervisor command, the use of LOGIN is

```
LOGIN <disk-name>
```

To use the LOGIN primitive to change a disk in a drive, first change the disk and then issue the appropriate LOGIN command.

A note of warning: if you issue a LOGIN command for a disk drive which does not exist, CP/M tells you in a message of the form:

BDOS Error: select on drive M

and then aborts micro-PROLOG. Your workspace program will have been lost and if it has not been saved you have some retyping to do!

#### 7.5.2 ERA

(ERA <file-name>) (erase the named file from its disk)

The effect of this unary relation/micro-PROLOG command is exactly as though the CP/M ERA <file-name> command had been typed under CP/M with the single exception that the <file-name> can be a micro-PROLOG accepted abbreviation for a .LOG file (see Chapter 3).

The ERA primitive succeeds if it successfully erased a file from the disk; if the file is not present on the disk or if the disk is marked read-only then the ERA call fails.

As a supervisor command, the use is

ERA <file-name>

#### 7.5.3 REN

(REN <file-name1> <file-name2>)

{change <file-name1> to <file-name2>}

The REN primitive renames the disk file with the name <file-name1> to the name <file-name2>. If a disk drive specifier is included with either file name then the file on that disk is renamed. If both <file-name1> and <file-name2> specify a disk letter they must both be the same.

If <file-name1> is not present on its disk then the REN primitive fails.

Since it is a two argument relation, REN cannot be used directly as a supervisor command. You must either use the ? query,

?((REN <file-name1> <file-name2>))

or define your own unary ren relation that reads in the second file name before call REN. (See Chapter 3.)

#### 7.5.4 DIR

(DIR <ambiguous-file-name> x)

{x the list of file x names matching the ambiguous name}

x must be a variable at time of call

The DIR primitive returns a directory in its second argument. The directory is a list of constants which are the file names that match the ambiguous file name given as the first argument. The ambiguous file name uses similar conventions to that of CP/M; all files that match the specifier will be returned in the list.

An ambiguous name of \* names all the .LOG files on the current disk, and \*.\* names all the files on the disk. A drive letter can be specified with the ambiguous name causing a directory of the given disk to be searched. The ambiguous file name "A:PROG?.LOG" names all the files on A that are .LOG files with a name PROG followed by at most one other character.

DIR is unusual in that the directory is returned as a value rather than printed on the console. The normal behaviour in CP/M is just to print it; however returning it as a value allows a micro-PROLOG program to manage its disks if required. For example, all the files of a given name can be searched by a micro-PROLOG program for some property.

We can use DIR to define a command that will display all the matched file names. The following program defines the dir command of MICRO.

```
((dir x)
  (DIR x y)
  (PP y))
{display list of file names matching x}
{find the list}
{pretty print it}
```

The command dir \* at the supervisor level of micro-PROLOG will cause a list of all the .LOG files on the default disk to be displayed.

#### 7.6 Record Input/Output

The two primitives FREAD and FWRITE allow items to be read and written in files in a fixed length record format. Using these two primitives a simple record file data base can be constructed and accessed from micro-PROLOG.

An argument to each of these primitives is the format specifier. A format is just a list of two element lists, each pair represents a field in the record. A field is either a numeric field or a constant field.

Numeric fields are signalled by the specifier:

(NUM n)

where n is a small integer in the range 1 to 15; this number is the field width: the number of ASCII characters in the record allocated to this field.

A constant field is signalled by:

(CON n)

where n is the length of the field as a number of characters. Any characters not used in a field are padded with trailing ASCII nul (0) characters; and if a particular field overflows the width of the excess characters are simply truncated; the maximum length of a CON field is 60 characters (the maximum length of a constant).

The total length of the record is given by the total of all the field widths in the format. This length must be greater than 0, and less than 255.



The following format specifies a record of four fields, the total length of the record is 135 characters:

```
((CON 30)(CON 30)(NUM 15)(CON 60))
```

#### 7.6.1 FWRITE

```
(FWRITE <file-name> <format> <list of constants & numbers>)
```

{write the given list of values, to the specified format, on named file}

The FWRITE primitive writes a single record as specified by the format with contents the list of constants and numbers. The record consists of a sequence of ASCII characters of length the total length of the format specifier. The file must have been opened with a CREATE.

The <format> is a list of field specifiers as outlined above; and the <list of constants & numbers> is just that.

The FWRITE call leaves the file pointer immediately after the written record. So the next written record will be adjacent to this one. Before the FWRITE you can position the file using SEEK. But make sure you position it at the beginning of the character sequence for a record.

Using SEEK and FWRITE in combination you can selectively update a file of records, overwriting an old record with new field values. But remember that the updated file must be one that has been opened for writing with a CREATE. So you cannot update an old file. To achieve the effect of updating an old file, open a new file for writing, and open the old file for reading. Then transfer all the records from the old file to the new one using a program such as

```
((copyfile X Y)
 (FORALL ((FREAD X <format> Z)) ((FWRITE Y <format> Z))))
```

You can now update the copied file of records using SEEK and FWRITE.

The data list must have the same number of elements as there are fields in the format: for each field in the format there must be a corresponding element in the list. The type of the field and the type of the corresponding element in the data list must also agree: if a constant is specified in the format then the corresponding element of the list must also be a constant. If the corresponding element is an unbound variable, then the "Control error" error is signalled. If the corresponding data element is given, but is of the wrong type, the FWRITE call fails.

The FWRITE primitive can also be used when writing to one of the 'special' files, including the console. In this case fields are padded not with trailing nuls (ASCII zero), but with spaces. Using FWRITE to the console is a simple way of generating formatted displays on the screen.

In the following call we write to the file F00.LOG a record consisting of a constant (20 characters) and a number (10 characters):

```
(FWRITE foo ((CON 20) (NUM 10)) (Marriene 34512))
```

Note that for numeric fields you should count enough spaces to allow for sign characters and the full E notation for larger numbers.

The call

```
(FWRITE "CON:" ((NUM 10)(NUM 10)(NUM 10)) (45.7 895.8 45.6))
```

displays the numbers, left justified, in fields 10 characters wide across the screen. It should be followed by a (PP) to position the cursor at the beginning of the next line.

#### 7.6.2 FREAD

```
(FREAD <file-name> <format> <list of vars, constants & numbers>)
```

{search for a record of the specified format that matches the data list of variables, numbers and constants}

The FREAD primitive is the counterpart to FWRITE. It enables records which have been written to be read back in. Apart from simply reading in a record, FREAD also searches for a record which matches the given constants and numbers in the data list of the call.

The constants and numbers in the records that are skipped before a match is found do not enter the micro-PROLOG data space. So a search down a very large file will not incur a memory turn over and garbage collection overhead. In contrast, the RPRED relation defined in 7.4.8 will turn over the memory when it is used to find a matching clause. But it is more general than FREAD. It searches a file of arbitrary clauses not just records, and it uses unification to find a match.

The relationship between the <format> and the data list is the same for FREAD as it is for FWRITE; except that variables are allowed in the list. When a matching record is found in the file the variables become instantiated with the values found in the corresponding fields in the record.

The FREAD call will fail if no matching record is found starting from the current file position or if there are not enough characters left in the file to fill a record. A successful call leaves the file at the character immediately following the found record. If SEEK is used to position the file before an FREAD you must make sure that you position it at the beginning of a record.

In the following call the file F00.LOG is searched (from the current position) for a record whose first component is Marriene:

```
(FREAD foo ((CON 20)(NUM 10)) (Marriene x))
```

If the next record that has Marriene in the first field is the one written by the example FWRITE call above, the call will succeed when the record is found with the variable x becoming bound to the number 34512.

FREAD can be used to read from the special files, such as the console.

The following program will read a single character at a time from the file x into variable y:

```
((one-char x y)
 (FREAD x ((CON 1)) (y)))
```

The call (one-char "CON:" y) therefore reads the next character typed at the console. You can use this to pick up a typed <return> by comparing the returned value of y with the ASCII code for <return>.

FREAD can also be used to skip characters on a file. For example, comments in a program source file can be skipped using this feature. In Pascal, comments are started by the character {, and ended by }. We have also made use of this convention in giving the listing of RPRED above. The following program reads the next term from a file x but it ignores { and } and any terms that are sandwiched between these brackets. In other words, if the next term is { it skips down the file until it finds a } and again tries to read the next term.

```
((read x y) {to read a term y from x ignoring comments})
  (READ x z) {READ the next term z}
  (decomment x z y) {y is z or next read-in term after
                    "}" if z is "{"}

((decomment x {" y) {READ term was "{" beginning a comment}
  (FREAD x ((CON 1)) {"}) {Skip to end of comment}
  (read x y) {y is next read-in term ignoring comments}

((decomment x y y) {next term is the READ-in term}
  (NOT EQ y {"}) {if this is not "{"}
```

If you develop micro-PROLOG programs using some file text editor, or you use such an editor on the files of SAVED programs, you can add comments using this convention. Of course, you cannot now load in the program using the primitive LOAD because this expects a program file without comments. You will have to write your own load making use of the above read.

The following is a suitable LOADC program for loading programs from files with comments. The non-comment terms in the file are the clauses of the program. Note: LOADC does not handle commented modules and bracketed comments can only appear between clauses. The comments cannot be intermingled with the text of the clause in the manner of the comments on the LOADC program.

```
((LOADC x) {load program file x ignoring comments}
  (OPEN x) {open the file for reading}
  (OR {fork to:}
    ((loadc x)) {load the opened file until EOF failure}
    ((CLOSE x))) {then close the file}

((loadc x) {load opened file x ignoring comments}
  (read x y) {find next non-comment term y}
  (ADDCL y) {add it as a clause to program}
  (loadc x) {continue with the load})
```

Note the use of the primitive OR described below to allow a recovery on the failure of loadc at the end of file. This failure on attempting to read beyond the end of file is how the tail recursive loadc evaluation terminates. The resultant backtracking on the loadc call will not undo the adding of all the read-in clauses. The data base manipulation primitives ADDCL and DELCL are never undone on backtracking.

Incidentally, the above definition of LOADC is very similar to the definition of LOAD embedded in the supervisor. If you do

a LIST LOAD you will get

```
((LOAD X)
  (OPEN X)
  (OR ((L* X) ((CLOSE X))))
```

You will not be able to LIST the L\* relation because it is not an available primitive. You will sometimes see relation names like this, usually just a \*, when you LIST supervisor programs. The \*s denote auxiliary relations that are not public. In the case of LOAD the L\* relation has a definition very similar to loadc. The difference is that loadc uses the comment skipping read whereas L\* uses the primitive READ.

### 7.6.3 Relations defined by files of records

As a final example of the use of FREAD we give an example of the definition of a micro-PROLOG relation in terms of a file of formatted records. Suppose that SALARIES.FOR is a file of records all of the format ((CON 20)(NUM 6)) which give employees' names and salaries. We can define a salary relation that will search this file whenever the relation is called and the file is OPEN. The definition makes use of an FPRED analogue of the RPRED relation defined in 7.4.8.

```
((salary x y)
  (FPRED "SALARIES.FOR" (010) ((CON 20)(NUM 6)) (x y)))

((FPRED X x Y Z) {find a record with format Y starting at
                  x on file X that matches Z}
  (SEEK X x) {position the file at x}
  (COPY Z Z1) {copy the template Z to Z1 replacing
               variables}
  (FREAD X Y Z1) {find a record matching the copy template}
  (SEEK X y) {find new file position}
  (OR ((EQ Z Z1)) {either unify Z with found record Z1}
    ((FPRED X y Y Z))) {or search for a matching record
                        from new position y}

((COPY () ()) {empty list is copy of empty template}

((COPY (x1y) (x1y1)) {copy starts with a new variable x1}
  (VAR x) {if template starts with a variable x}
  (COPY y y1)) {followed by a copy of the tail}

((COPY (x1y) (x1y1)) {copy starts with x}
  (NOT VAR x) {if this is not a variable}
  (COPY y y1)) {followed by a copy of the tail}
```

The copying of the given pair of arguments of the call to salary is needed in order that the values found for any variables in the call, and returned by the (EQ Z Z1) condition in FPRED, are undone when there is backtracking to find an alternative solution to the call. When backtracking causes the 'else' branch of the OR to be taken the effect of the EQ is undone and new values for the variables in Z are sought by the tail recursive FPRED call.

## 7.6.4 Mixing records and terms

There is nothing to stop you writing a file using one format and reading it with another - but beware, at least the field widths should be the same because of the packing 'nuls'. You can also write part of a file with one format, part with another. You should keep track of where one group of formatted records ends and the other begins by using `SEEK` to find the file position when you have finished writing with one format. The formats are, after all, just logical concepts superimposed on a segment of contiguous characters.

In the same way you can mix records with terms. The records are written and read using `FWRITE` and `FREAD` and the terms using `READ` and `W`. A term on the file is just a sequence of characters satisfying the syntactic constraints of `micro-PROLOG`. You must be very careful with this mix. A fairly safe mixture is to have the first written thing on the file be the list term that specifies the format for the records on the rest of the file. This can be read using `READ` and then used in the subsequent `FREADs`. This is a useful device if you want to program a general purpose formatted file handling utility.

## 7.7 Type Predicates

The type predicates test a single argument for its type: i.e. whether it is a number, constant, list or unbound variable.

7.7.1 `NUM`

`(NUM x)` {x is a number}

The `NUM` built-in predicate tests to see if its single argument is numeric or not. If it is a number the call succeeds, if it is an unbound variable, a constant or a list then the call fails. For example, `(NUM 3)` and `(NUM -1.3e9)` succeed (are true), whereas `(NUM bill)` and `(NUM x)`, with `x` unbound, fail (are false).

7.7.2 `INT`

`(INT x)` {x is an integer}

The `INT` predicate, in its single argument form, checks its argument for being an integer. A number is classed as an integer even if it is a large integer that has to be represented as a floating point number. Hence, `(INT 3.45e9)` is true. The call fails if the argument is an unbound variable or it is a non-integer value.

7.7.3 `CON`

`(CON x)` {x is a constant}

The `CON` built-in predicate tests if its single argument is a constant. For example, `(CON foo)` succeeds, whereas `(CON ())`, and `(CON 1)` both fail. If the argument is not a constant (including the case where it is a variable) then the call fails.

7.7.4 `LST`

`(LST x)` {x is a list}

The `LST` predicate is true of lists. This includes the empty list. If the single argument is not a list, (again including the case where it is an unbound variable), the call fails.

7.7.5 `SYS`

`(SYS t)` {t is an atom for a primitive relation}

`SYS` tests to see if its argument is a call to a built-in relation (i.e. one of the relations described in this chapter). It succeeds if it is, it fails if it is an atom for any other relation.

7.7.6 `VAR`

`(VAR x)` {at the time of the call x is an unbound variable}

Strictly non-logical, the `VAR` built-in type predicate checks to see if its argument is currently a variable. (It is non-logical because a successful call is invalidated if the variable is subsequently bound.)

## 7.8 Logical operators

The basic clausal form of logic programs, which is limited to the implicit 'and' between the calls in the body of a clause, is extended via the logical operator primitives of `micro-PROLOG`. These are supervisor programs that implement: disjunction `OR`, negation-as-failure `NOT`, conditional alternatives `IF`, list of all solutions to a query `ISALL`, test on all the solutions to a query `FORALL`, identity `EQ` and the restriction to a single solution `!`. These can be used to increase the efficiency and the readability of `micro-PROLOG` programs. They also raise the level of the language, making it more powerful and expressive.

7.8.1 `OR`

`(OR <atom list1> <atom list2>)`

{either the query ?<atom list1>  
or the query ?<atom list2> can be solved (is true)}

The disjunctive operator `OR` has two arguments: each of which is a list of atoms. An `OR` call succeeds if either of its component queries is solved. For example,

`(OR ((father-of x y)) ((mother-of x y)))`

succeeds if either `(father-of x y)` or `(mother-of x y)` succeeds (is true).

An empty goal (named by the empty list `()`) always succeeds, and hence if used as an argument to `OR` acts as a 'true' branch. The supervisor definition of `OR` is

`((OR X Y)|X)`

```
((OR X Y)|Y)
```

This uses the 'meta-variable as the rest of the clause' form (see Chapter 2). From the definition we can see that the 'either' branch is tried first (it is the first clause). Only when there are no more solutions to this branch will a solution to the 'else' branch be sought. As we saw in the example program of 7.6.2, the else branch can be used to specify the failure alternative to the 'either' branch. A / placed in the 'either' branch will cut out the 'else' branch option when it is evaluated. Thus, the / in

```
((OR ((test X)/(process X Y)) ((default X Y)))
```

will cut out the 'else' branch if (test X) succeeds. See below for more information on /. A / in any of the logical operators only has an effect on the backtracking within the operator. It has no effect on the backtracking evaluation of the other conditions in the clause or query in which the logical operator appears.

#### 7.8.2 NOT

```
((NOT <relation-name> t1 t2 .. tk) (A)
```

{the condition (<relation-name t1 t2 .. tk) fails (is false)}

The NOT operator implements negation-as-failure [Clark 1978]. It denotes the negation of the call

```
(<relation-name> t1 t2 .. tk) (B)
```

The negation (A) succeeds if, and only if, the unnegated call (B) fails.

There is an implicit assumption that the program definition of the relation is complete; that all the positive instances can be inferred from the program definition. On this assumption, it is correct to assume that a condition is false (hence that its negation is true) if an attempt to prove the condition fails.

A negated condition can only be used for testing, it cannot be used to find any values for variables in the condition such that the condition is false. Indeed, the more correct reading of the call is

```
show that
there are no x1,...,xn such that
(<relation-name> t1 .. tk)
```

where the x1,...,xn are all the variables in the call at the time that it is evaluated. This means that the reading and effect of a program which uses NOT can be effected by the position of the NOT.

As an example,

```
?((parent-of John x)(NOT male x))
```

is read: show that John has a child who is not (provable) male

Whereas

```
?((NOT male x)(parent-of John x))
```

must be read:

```
show that
there is no x such that x is male and
show that there is an x who is a child of John
```

This reading is forced because the x in (NOT male x) will be unbound when the condition is evaluated. The (NOT male x) condition will fail if there is at least one way of showing that the is a male.

The fact that NOT, implemented as 'failure to prove', is only an approximation to the logical negation is evident in the use of a double NOT. The queries:

```
?((parent-of tom X)(PP X))
?((NOT NOT parent-of tom X)(PP X))
```

are logically equivalent. However, with the first, the name of any found child of tom will be printed, with the second the unbound variable X will be printed. The double NOT succeeds if (parent-of tom X) succeeds but it will leave the variable X unbound. The first query is read:

```
show that
there is an X such that (parent-of tom X)
and display the found X
```

The second is read:

```
show that
there is an X such that (parent-of tom X) and
then show that there is an X that can be displayed
```

The second reading derives from the fact that the X in the negated condition is unbound when it is evaluated.

The positive aspect of this imperfection of NOT is that double negations can be used to test if a condition succeeds without binding any variable in the condition.

**RULE OF THUMB:** Place negated, test conditions on a variable (or variables) after positive conditions/calls that can be used to find values for the variable(s).

The supervisor definition of NOT is

```
((NOT I X) X
 / FAIL)
((NOT I X))
{If X is provable then (NOT I X) must fail}
{else, (NOT I X) is proven}
```

Note the essential use of / to prevent the use of the second clause if X succeeds. It also makes sure that X is only proven once. Only when there is no proof of X will the second clause be used. This confirms (NOT X) without instantiating any variables - hence the restriction of NOT to test use.

## 7.8.3 IF

```
((IF <atom> <atom list1> <atom list2>))
```

{either <atom> and <atom list1> can be solved (are true) or (NOT ! <atom>) and <atom list2> can be solved (are true)}

The IF is a conditional alternative. Its use is equivalent to

```
((OR ((<atom> / <atom list1>)) ((atom list2)))
```

which uses / to prevent the use of the 'or' branch if the <atom> test is solved. The IF form is more declarative. The declarative equivalent using OR would be

```
((OR ((<atom> <atom list1>)) ((NOT ! <atom>)) <atom list2>))
```

with an explicit (NOT ! <atom>) on the 'else' branch. This form is much less efficient than the IF or the OR using / because of the repeated evaluation of <atom> in the second branch.

A definition of a relation R of the form

```
((R ...) (IF (P ...) (<bodyA>) (<bodyB>)))
```

is (almost) equivalent to a definition using the pair of clauses:

```
((R ...) (P ...) <bodyA>))  
((R ...) (NOT P ...) <bodyB>))
```

The difference is that in the IF definition the test (P ...) is only ever evaluated once.

The use of conditionals in this way is a mixed blessing because less use can be made of unification. For example in the two clauses for R above it could be that the heads of the two clauses would naturally be slightly different. When the conditional form is used the head must be the 'most general' of the two, with extra equalities in the conditional branches to bind the variables in the head to the terms that they should be. For example,

```
((sort (x y|Z) (x |Z1))  
 (LESS x y)  
 (sort (y|Z) Z1))  
 ((sort (x y|Z) (y |Z1))  
 (NOT LESS x y)  
 (sort (x|Z) Z1))
```

must be absorbed into

```
((sort (x y|Z) Z2)  
 (IF (LESS x y)  
 ((EQ Z2 (x|Z1)) (sort (y|Z) Z1))  
 ((EQ Z2 (y|Z1)) (sort (x|Z) Z1))))
```

which is much less readable. A non-declarative solution is to use / after the test of the first clause and to drop the test in the second.

```
((sort (x y|Z) (x|Z1)) (LESS x y) / (sort (y|Z) Z1))
```

```
((sort (x y|Z) (y|Z1)) (sort (x|Z) Z1))
```

The supervisor definition of IF uses / in just this way. It is:

```
((IF X Y Z) X / ! Y)  
((IF X Y Z) ! Z)
```

As with OR, a / placed inside the 'then' or the 'else' branch of an IF only effects the backtracking within that branch.

## 7.8.4 EQ

```
((EQ t1 t2) (t1 is identical to t2))
```

The supervisor definition of EQ is

```
((EQ X X))
```

so the effect of the evaluation of an EQ condition is the attempted unification of its two argument terms.

```
((EQ (x1 x2) (A B)) results in x1=A, x2=B.  
((EQ (a|z) (x y c)) results in x=a, z=(y c).)
```

## 7.8.5 ?

```
((? <atom list>) (true if the sequence of atoms in <atom list>  
 can be solved (are all true)))
```

The supervisor definition of ? is

```
((? x)|x)
```

? can be used when the syntax for a condition requires a single atom but you want to 'pass' a 'conjunction' of atoms. An example is the IF condition of which the test must be an atom. The form,

```
((IF (? <atom list>) (...) (...))
```

enables you to have several conditions for the test.

As a supervisor command ? is the primitive query form of micro-PROLOG. The more elaborate query forms of MICRO and SIMPLE are all defined using ?. For example, the following is the definition of the which of MICRO:

```
((which (X|Y)) {X is the answer pattern, Y the query}  
 (? Y) {solve Y}  
 (PP X) {display answer pattern X for this solution}  
 FAIL) {backtrack to find next solution}  
  
((which (X|Y)) (when first clause fails)  
 (PP No (more) answers)) {there are no more answers to Y}
```

## 7.8.6 FORALL

```
(FORALL <atom list1> <atom list2>)
```

{for all the solutions of <atom list1>, <atom list2> can be solved}

FORALL is a very high level concept, and can often replace explicit recursions in a program.

The following program defines 'prime number' using FORALL. It is a specification that can be used as a prime checking program.

A positive prime number  $x$  is a number such that none of the integers  $y$  in the range  $2 \leq y < x$  divide  $x$ . That is, for each  $y$  in this range it is not the case that  $y$  divides  $x$ . This definition is formalised as:

```
((prime x)
 (FORALL ((in-range 2 x y))
  ((NOT divides y x))))
```

where in-range and divides are:

```
((in-range x y x))
((in-range x y z)
 (SUM x 1 x1)
 (LESS x1 y)
 (in-range x1 y z))
```

```
((divides x y)
 (TIMES x z y)
 (INT z))
```

This program is not a very efficient program for checking prime numbers, but it is an obviously correct one.

A (FORALL <atom list1> <atom list2>) condition is solved if, and only if, the pair of conditions:

```
(? <atom list1>) (NOT ? <atom list2>)
```

does not have a solution. In other words, if there is no way of solving <atom list1> so that <atom list2> cannot be solved. Hence, the supervisor definition of FORALL is

```
((FORALL X Y) (NOT ? ((? X)(NOT ? Y))))
```

After the evaluation of a FORALL all variables in the <atom list> arguments of the condition are left unbound.

## 7.8.7 ISALL

```
(ISALL t t1 <atom1> <atom2> .. <atomk>) k>0
```

{t is the list of all the t1's such that the query ?(<atom1> ... <atomk>) is solved}

ISALL can only be used to instantiate variables in t

Each element in the list t is a copy of the value the term t1 is given by each different solution to the query. At the end

of the evaluation all variables in t1 and the <atom> conditions are left unbound.

The solutions found are neither unique nor sorted: if a different solution to the query results in the same value of t1 then a second copy of this value appears on the list t. The values for t1 appear on t in the reverse of the order in which the solutions to the query condition are found.

The first value of t1 on t corresponds to the last solution of the query, the last value of t1 on t corresponds to the first solution of the query. t is usually given as a variable in the call and the evaluation binds the variable to the list of solutions. However, t can be a list or a list pattern. If it is a list, the elements on t must be in the order that they would be found by an evaluation in which t was given as a variable, which is the reverse of the order in which solutions to the query are found. Since this order is not easy to predict, this checking role for ISALL is not very useful.

Example uses of ISALL are:

```
(ISALL x y (father-of tom y)(male y))
```

{makes x the list of all the sons of tom in the reverse of the order they are found}

```
(ISALL (x|y) (z1 z2) (gives bill z1 z2))
```

{checks that bill gives at least one thing z1 to someone z2 and makes x the last (z1 z2) pair found, y the other solutions in reverse order}

```
(ISALL (x1 y1) y (mother-of mary y))
```

{checks that mary has exactly two children, and finds there names in the list (x1 y1)}

```
(ISALL (bill) y (father-of tom y))
```

{checks that bill is the only child of tom}

ISALL is not defined entirely as a micro-PROLOG program. The following definition approximates the built-in definition:

```
((ISALL X Y|Z)
 (init x ())
 (FORALL ((? Z)) ((update x Y)))
 (DELCL ((value x X))))
 {initialise value assertion for x to ()}
 {for all the solutions to (? Z) update the value assertion for x with the value of Y}
 {X is the final value of x}
```

```
((init x Y)
 (DELCL ((index x))))
 {find and delete index assertion to get}
 {an index value for x}
 (SUM x 1 Z)
 (ADDCL ((index Z)))
 {add new index assertion}
 (ADDCL ((value x Y)))
 {add initial value of x assertion}
```

```
((update x Y)
 (DELCL ((value x Z)))
 {find and delete old value assertion for x})
```

```

(ADDCL ((value x (Y|Z)))) {add new value assertion for
  x with added Y}

((index 1))
  {initial index assertion}

```

The use of the index assertions allows nested ISALLs.

The built-in definition of ISALL does not use the relatively slow additions and deletions to the data base to keep track of the partial list of solutions to the query. It directly updates a list of partial solutions. Its evaluation is very fast. The time taken to find the list of solutions is only a little longer than the time taken to search for all the solutions to the query condition.

You can use ISALL to define a count relation that counts the number of different solutions to a query. The following definition, which uses a tail recursive list-count, will take little longer than the time to search together with the time to count to the number of found solutions:

```

((count X Y|Z)
 (ISALL X1 Y|Z)
 (list-count X1 0 X))

((list-count () Y Y))
((list-count (X|Y) Y Z)
 (SUM Y 1 Y1)
 (list-count Y Y1 Z))

```

## 7.8.8 L

```

(! <relation name> t1 .. tk)
{the first solution to <relation name> t1..tk}

```

The evaluation of the ! call reduces to the evaluation of the atom that follows the ! in the call. However, it restricts the evaluation to just one solution. On backtracking, no further ways of solving the condition are sought. The call

```
(! likes x tom)
```

should be read "the first x who likes tom".

! provides useful control information when you know that there will only be one way of solving the condition. It cuts out the redundant search on backtracking. It is particularly useful for test calls that will be evaluated against a large number of assertions. In the MICRO which query

```
which(x (parent-of tom x)(male x))
```

each found child of tom is checked for being male. The backtracking search results in alternative and redundant 'proofs' of the (male x) condition being sought before another child of tom is found. In consequence, the entire set of male assertions will be scanned for each found child. By using

```
which(x (parent-of tom x)(! male x))
```

the scan of the male assertions is abandoned as soon as the found

x is confirmed as a male.  
The supervisor definition of ! is:

```
((! ! X) X /)
```

It is the / that prevents the search for a second solution.

## 7.9 Data base operations

micro-PROLOG has three primitives that enable clauses to be accessed, added and deleted from the user's workspace at run-time. The ability to add and delete clauses is needed in order to implement extensions to the supervisor. It also enables the data base to be used as a scratch pad memory. An example of this latter use is the definition of ISALL given above. Being able to pick up the clauses for a relation allows the definition of an alternative query evaluation strategies. An example of an alternative evaluator is given below.

## 7.9.1 CL

This has two forms of use, a single argument form and a three argument form:

```

(CL X)
{X is a clause in the program}

(CL X Y Z)
{X is a clause at position Z in the
sequence of clauses for its
relation with Z >= Y}

```

At the time of evaluation of either call X must be a clause or a clause pattern in which at least the relation name of the head atom is given as a constant. For the three argument form Y must also be given as a positive integer, Z can be a variable or be given.

This program accesses clauses from the user's workspace (or the currently opened module). It is one of the few built-in programs that is at all non-deterministic as it can be used to backtrack through an entire relation. The important constraint on CL is that the relation name of the head atom of the clause to be found must be given.

For example,

```
(CL ((Atlix)|X))
```

succeeds if there are any At clauses in the workspace so this form of use can be used to test if a relation is defined before a call to the relation is made. (A call to an undefined relation signals an error.)

In the example call, if there are clauses for At, variable x is bound to the arguments of the head atom of the first clause, and the variable X is bound to the (possibly empty) list of atoms that make up the body of the clause. Backtracking will result in an alternative (later) clause being sought for the At relation. For the most general use, the clause argument is a pattern of the form

```
((<relation name>|Y)|Z)
```



The three argument form of CL can be used to find particular clauses in the program, the second argument must be given. It gives the position of the clause from which the search commences. The last argument is the position of the found clause.

```
(CL ((likes John X) Y) 1 X)
```

binds X to the position of the clause that matches ((likes John X) Y) with the search starting at the first clause. Backtracking results in an alternative clause being sought that matches the clause pattern. If there is one, its (later) position will be given as the value of X and the rest of its structure will be given in the bindings for x and y.

We can also use the three argument form to pick up a specific clause. To do this, we use the most general pattern for a clause for the relation and give the clause position.

```
(CL ((likes(x) y) 4 4)
```

will bind x and y to the argument list and the body of the fourth clause for likes if it exists. If it does not, the call fails.

The single argument CL behaves almost as if a micro-PROLOG program is stored as a sequence of CL clauses with the actual program clauses as argument terms. The evaluation of a CL call is then a search through these CL clauses. This is almost true. The sequence of CL clauses is implicitly represented by the relation names in the dictionary which 'point' to the sequence of clauses that define them.

However the restriction that the predicate symbol must be known at the time of the call, means that CL can only be used to search through the clauses of a **single** relation, it cannot be used to search through all the clauses in the workspace.

CL can be used to define a query evaluator.

```
((question ())) {a question with no conditions is true}

((question (X|Y)) {a non-empty question is true}
 (CL (X|Z)) {if there is a clause (X|Z) matching X}
 (question Z) {whose body Z is true}
 (question Y)) {and the remaining conditions Y are true}
```

This makes use of CL to find a clause that solves X in such a way that the rest of the question can be solved. Failure to solve Y will first result in backtracking on the evaluation of the body Z, and finally on the search for an alternative matching clause. For questions and programs that do not use /, it is equivalent to the supervisor ?.

By collecting all the matching clauses as a list, we can try the clauses in an order different from their order in the program. An alternative recursive clause for question is:

```
((question (X|Y))
 (ISALL z (X|Z1) (CL (X|Z1))) {z are the clauses matching X}
 (select (X|Z) z) {(X|Z) is a selected clause}
 (question Z)
 (question Y))
```

The definition of select determines the order in which clauses for the condition X are tried. As an example,

```
((select x z)
 (sort z z1)
 (member-of z z1))
```

can be used to select the clauses in order of increasing number of atoms in the body given suitable definitions of sort and member-of.

Elaborations of this approach enable one to program breadth first evaluation of queries (as distinct from depth first with backtracking).

### 7.9.2 ADDCL

This also has two uses:

```
(ADDCL X) {add X as a new last clause for its relation}

(ADDCL X <int>) {add X as a clause after clause
 <int> for its relation}
```

For both uses, at the time of evaluation X must be a list term that satisfies the syntax of a clause. In particular the relation name of the head atom must be given as a constant. For the second use, <int> must be a non-negative integer.

The clause added is a copy of the list term X in which any unbound variables of the term become variables in the added clause. This convention, that unbound variables in the 'name' X denote variables in the clause to be added is logically not very satisfactory. However, all PROLOG implementations use this convention. It enables clauses to be read in using the R primitive and then added to the program using ADDCL.

R converts variable names in the read-in clause term into internal form unbound variables. ADDCL then copies the clause term, mapping the unbound variables into special variable names in the added clause. (This R, ADDCL cycle is how the supervisor accepts new program clauses - see the definition of <SUP> below.)

When a clause is picked up by the micro-PROLOG interpreter during an evaluation, or when it is retrieved using CL, the special variable names of the clause are converted back into entirely new internal form variables, just as though the clause had been read-in.

So each use of the clause gets a fresh copy, with a new set of variables different from any other variables currently in the evaluation. This allocation of new internal form variables (which are actually locations where pointers to values for the variables will be stored when the variables are bound) for the variable names of an accessed clause is exactly the same as the allocation of new locations for the variables of a procedure that takes place in a conventional recursive programming language.

If ADDCL is used with a single argument then the clause is added to the end of the appropriate program. Otherwise it is inserted after the clause whose position is given as the second argument. If 0 is used as the clause number, the clause is inserted at the front of the program.

```
(ADDCL ((append () x x)) 0)
```

adds the clause

```
((append () x x))
```

to the front of the append program. If a position is given that is beyond the last clause for the relation the clause is added as a new last clause.

RESTRICTION: clauses cannot be added for primitive relations nor for relations exported by some loaded module. An attempt to do so signals an error.

### 7.9.3 DELCL

```
(DELCL X)      (delete the first clause matching X)
(DELCL <relation name> <int>) (delete the <int>th clause
                             for <relation name>)
```

X must satisfy the syntax of a clause, in particular, the relation name of the head atom must be given.

The primitive DELCL is used to delete clauses from the workspace (or currently opened module). In the first form the program is searched for a clause matching X. The first one found is deleted. If none is found, the call fails. In the second form the clause to be deleted is specified by a relation name and a clause position. Again, if the specified clause does not exist, the call fails.

```
(DELCL (append | x1 | x2))
```

will cause the first clause for append (if there is one) to be deleted. However, it will also bind the variables x1 and x2 to the head arguments and body of the clause respectively. So this form of DELCL can retrieve information from the deleted clause. An example of this use is in the definition of ISALL given above. In particular, the definition of the auxiliary relation update uses ADDCL and DELCL to manipulate the data base as a scratch pad memory.

RESTRICTION: clauses for primitive relations or relations exported from loaded modules cannot be deleted. An attempt to do so signals an error.

### 7.9.4 Undoing the effect of ADDCL or DELCL

The effect of an ADDCL or a DELCL is not undone on backtracking over the call. The adding or deleting of a clause is a side effect on the state of the program in the same way that R and PP are a side-effect on the state of the terminal. We can never undo the effect of a R or PP, but we can undo the effect of ADDCL or DELCL - by doing the opposite.

The following clauses define 'soft' addcl and delcl relations whose effect is undone on backtracking. This is subject to the proviso that a / has not been evaluated after the call that cuts out their 'else' branches.

```
((addcl X)
 (OR ((ADDCL X)) ((DELCL X))))
((delcl X)
 (OR ((DELCL X)) ((ADDCL X))))
```

Both relations need the clause to be unambiguously specified - to be uniquely determined by the argument X. If it is not, the DELCL of the addcl definition may delete a different clause that happens to match X, and the ADDCL of the delcl definition may add a more general clause than the one deleted.

Notice that delcl may not add the clause back in the same position so it is only useful when the position of the clause is not important. This is usually the case when you are manipulating a data base of single atom clauses.

### 7.9.5 KILL

```
(KILL R)      (delete all clauses for relation R)
(KILL (R1 .. Rk)) (delete all clauses for each of R1..Rk)
(KILL ALL)    (delete all clauses from workspace or
               all those owned by the currently opened
               module)
(KILL <module name>) (delete the named module)

R and R1 .. Rk must be relation names.
```

The KILL primitive deletes all clauses for a single relation, all those for a list of relations, or all the clauses that can be deleted. When working in the workspace (the supervisor prompt is &.) only user relations in the workspace can be KILLED. A KILL ALL will delete all the clauses from the workspace but will not delete any loaded module. This must be deleted with the last form of use KILL <module name>.

When working in an opened module (see below) KILL can be used to delete all clauses for relations that are owned by the module. The main use of KILL is as a supervisor command.

### 7.10 Library procedures

These allow saving of programs, loading programs and listing programs at the console.

#### 7.10.1 LIST

```
(LIST R)      (list program for relation R)
(LIST (R1 .. Rk)) (list program for each of R1 .. Rk)
(LIST ALL)    (list programs for all workspace
               relations or all relations
               currently opened module)
(LIST <module name>) (list the named module)
```

R R1 .. Rk must be relation names

The LIST primitive lists the specified programs, in a standard format, at the console.

```
(LIST ALL)
```

lists the whole program, (apart from any loaded modules), and

```
(LIST (Likes Fred Angle))
```

lists the programs for Likes, Fred & Angle.

## 7. Built-in Programs

Modules are listed in the form in which they are SAVED - see below.

### 7.10.2 SAVE

```
(SAVE <file name>)
  (save entire program in named file)
(SAVE <file name> (R) .. RK)
  (save only programs for given relations)
(SAVE <file name> <module name>)
  (save named module in named file)
```

For the first form of use the entire program saved will be the workspace program if working in the workspace. If working in an opened module it will be all the clauses owned by the module. The supervisor command

```
SAVE "A:TEST.LOG"
```

saves the entire program on the disk file TEST.LOG on drive A. If there is already a file with that name, it is renamed with extension .BAK. This ensures automatic backing up of programs.

Programs are saved as a sequence of clauses in the same format in which they are displayed by LIST. Modules are saved in a special format - see the next section on modules. It is also the format in which modules are displayed by LIST.

The second two forms of use cannot be used directly as supervisor commands as they have two arguments. For a command to save a module, you must use a query of the form:

```
?((SAVE <file name> <module name>))
```

SAVE does not delete the saved program(s). It also automatically opens the file for writing and then closes it on successful completion.

**WARNING:** File names, relation names and module names must be distinct. If you inadvertently use a name for more than one of these roles you will get an error. If you attempt to save a program on a file, the file name you use must not be the same as any current relation name or any loaded module. You must be wary, especially when you use an allowed abbreviated name that does not give the drive or the .LOG extension (see Chapter 3).

The error can occur when you are loading a file with the LOAD command described below and the LOAD reaches a clause for a relation that has the name of a loaded module or a currently opened file. On the error the LOAD will be abandoned with the file left open. So you should CLOSE the file from which you were loading if this occurs. If you are using an error handler (see Appendix A) the offending relation name will appear in the ADDCL call that caused the error.

To avoid this type of error use different forms of name for relations, modules and files. Most of the micro-PROLOG modules on the distribution disk have a name of the form <name>-mod, where <NAME>.LOG is the file in which they are supplied. If you always use the .LOG form for the file name, and have module names ending with -mod, you should avoid the error.

## 7. Built-in Programs

### 7.10.3 LOAD

```
(LOAD <file name>) (load the program from named file)
```

This program reads the disk file, and adds all the clauses in the file to the workspace or the currently opened module. If the file contains a saved module, the clauses are loaded as a module and do not enter the workspace. Note: a module can only be loaded when working in the workspace (the supervisor prompt is &.). You cannot load a module when working in an opened module. The attempt to do so signals an error. You can only load into an opened module files consisting entirely of program clauses. The supervisor command

```
LOAD TRACE
```

loads the program in the disk file TRACE.LOG on the currently logged in disk. TRACE is an example of an abbreviated file name. micro-PROLOG converts all file names given without an extension into names with the .LOG extension.

As each clause on the file is read in, it is added to the end of the current program for its relation. If there are clauses for the relation before the load, all the loaded clauses will be added after the existing clauses.

LOAD automatically opens the file and then closes it on successful completion of the LOAD. See the warning given 7.10.2.

### 7.11 Module Construction Facilities

micro-PROLOG has facilities for constructing modules. These are named micro-PROLOG programs (sequences of clauses) which communicate with workspace programs and other modules via import/export name lists. Names used in the module that are not in the import/export lists are local to the module and are invisible from outside the module. Different modules can therefore use the same local names with no name clash. On the other hand, constants that do need to be communicated across the module **must** be in the import or export list.

A module has five components:

a **name** (which is a micro-PROLOG constant)

an **export list**: a list of constants which are being 'made available' outside the module. These are usually the names of relations of the module. An exported relation can be used in a workspace program or command as though it were a primitive relation when the module is loaded. An exported relation name can also be imported by another module and used by the other module as though it were a primitive.

an **import list**: a list of constants that the module imports from the outside. This must include the names of relations defined in the workspace or exported from some other module that need to be accessed from inside the module. It must also include all the 'data' constants used in the module that need to be communicated across the module e.g. constants which may be used in calls to exported relations of the module and must be recognised by the module or constants used in the module to calls to imported

relations which will need to be recognised outside the module. Such communicated 'data' constants can be given in either the import or export list, but it is good practice to restrict the export list to exported relation names.

a **local dictionary**: constants appearing in the module which are private to that module

the **module program**: all the clauses owned by the module. These are the programs for all the relation names of the export list and the local dictionary.

The export list, the import list, and the local dictionary have no names in common. The relations **accessible** by a module are the relations defined by the clauses it owns and the relations with names in the import list.

Modules are **LOADED** and **SAVED** automatically by the **LOAD** and **SAVE** programs. For the **LOAD** the same form of call is used, (**LOAD** <file name>), even if the file contains a module. This is possible because files containing modules have a different structure to ordinary program files and this is recognised by the **LOAD** which then handles the file in a special way. For the **SAVE** there is a special three argument form for modules:

```
(SAVE <file name> <module name>)
```

This saves the named module on the specified file in the form:

```
<module name>
<export list>
<import list>
```

```
.      (clauses owned by the module)
```

```
CLMOD
```

The local dictionary is not saved because this is automatically reconstructed by the **LOAD**.

**Warning:** if you use a text editor to develop a module program file the terminating **CLMOD** must be followed by a <return>.

The **LIST** program can also be used to list the contents of a module:

```
(LIST <module name>)
```

will display the module at the console in the form that it is saved on a file.

You can enter and exit loaded modules with the **OPMOD** and **CLMOD** commands described below. Entering a module makes it the **current** module.

The supervisor uses the current module's name as its prompt, so if the current module is called **Simple** then instead of the & prompt we get the prompt:

```
Simple.
```

The top-level prompt &. is in fact the name of the root module & which is also the workspace. The root module has a

special role. On loading **micro-PROLOG** the root module becomes the current module. It exports no names but it imports all the names exported by the loaded modules. As a module is loaded, the import list of & grows. Finally, other modules can only be loaded when & is the current module.

A supervisor **LIST ALL** will list all the clauses owned by the current module, and all clauses entered at the keyboard or added using **ADDCL** are added to the current module. An attempt to add or delete a clause for an imported name of the current module signals an error.

It is possible to have a program in a module which when called adds clauses to an imported relation of the module but the program cannot be called while the module is the current module. For example, the **SIMPLE** front end program imports the relation name **dict**. It maintains this relation for the user by adding and deleting clauses from it. It can do this because when the **dict** clauses are added and deleted the root module & is the current module and **dict** is not an imported relation of the workspace - it is a relation owned by the workspace.

The local dictionary of the current module is the dictionary used by all the I/O primitives of **micro-PROLOG**. When a constant is read-in, it is looked up in the local dictionary of the current module. If it is not present, it is added to the local dictionary. This means that any module program that is to be called from a workspace query, which reads in and tests for certain constants in the input, must import all the constant names it needs to recognise. For, when they are read-in, & will be the current module and the constants will enter the local dictionary of the workspace. The program in the module will not be able to recognise them unless it imports their names.

There are four primitives connected with modules: **CMOD**, **CRMOD**, **OPMOD** and **CLMOD**.

#### 7.11.1 CMOD

```
(CMOD x) {x is the name of current module}
```

x must be a variable at call

The **CMOD** program simply returns the name of the current module. It is used, for example, by the supervisor to print out the name of the module as part of its top-level prompt.

#### 7.11.2 CRMOD

```
(CRMOD <module name> <export list> <import list>)
```

{create an empty module and make it the current module}

**CRMOD** creates a new module with name <module name> and with the given export and import constant lists. It then enters it, i.e. makes it the current module. The <export list> and the <import list> are as defined above.

**CRMOD** can only be used when & is the current module. The <module name> cannot be the same as the name of any relation accessible by &. It must also be different from the name of any other current module and any opened file. Finally, none of the exported names can be the names of workspace relations or relations exported by other current modules. If any of these constraints are flaunted the "illegal use of modules" error is

raised.

Having created a module you can enter clauses into the module using all the facilities of the resident supervisor described in Chapter 3. But note that you will not be able to LOAD and use the structure editor (of Chapter 4) or any of the other program development utilities whilst in the new module. The only way you can use the editor to help develop or modify a program inside a created module is to import the name EDIT and all the names of the edit commands into the module. The editor must be loaded whilst at the root module (&) level, but it can be called from inside the new module.

A more convenient method of developing a program that is to be wrapped up in a module is to develop it first as a workspace program which is saved in the normal way. You then use CROMOD to create the shell for the new module. On entry, LOAD the saved workspace program, then exit the module with the CLMOD described below. You can now SAVE the new module using the special form for modules. Alternatively, use the MODULES utility described in Chapter 4 which supports the construction and modification of modules.

CROMOD is used by the LOAD primitive when it encounters a module name in the file. It creates a module using the name and the export and import lists that immediately follow the name. It then reads in all the clauses that follow up to the end of module mark CLMOD in the file. Each clause is added to the newly created module, which temporarily becomes the current module. On reaching the CLMOD the module is exited and the current module is again the & workspace module. If there is an error on loading, or you interrupt the load with a ^C, you will find that you are in the temporarily entered module.

#### 7.11.3 OPMOD

```
(OPMOD <module name>) {make named module the current module}
```

OPMOD enters the already existing named module and makes it the new current module.

#### 7.11.4 CLMOD

```
CLMOD {close current non-root module and return to &}
```

CLMOD drops out of the current module back into the root & module. It is not possible to drop out of the root module. As a supervisor command OPMOD must be given an (ignored) argument. e.g.

```
CLMOD.
```

with the . the ignored argument.

### 7.12 Miscellaneous Predicates

In this section we draw together a rag-bag of primitives not covered above. These include the dictionary relation and some control primitives.

#### 7.12.1 DICT

```
(DICT x y z | X) {y z and X are respectively, the export list, the import list, and the local dictionary of the current module x}
```

The DICT relation defines the dictionary of the current module. If you do a LIST DICT you will get a clause of the above form listed. You can also call DICT to pick up any of its arguments. Note that DICT is a multi-argument relation. After the import list z is a sequence X of all the local constants of the current module. Garbage collection will remove from X all constants no longer in use.

#### 7.12.2 QT

```
QT {quit micro-PROLOG and return to CP/M}
```

Execution of this program will cause an exit from the micro-PROLOG system into CP/M. This is the only recommended method of exiting the micro-PROLOG system. As a command it must be given an (ignored) argument.

#### 7.12.3 /

```
/ {always true but with a side effect on evaluation}
```

The / primitive is used to control backtracking. When executed as a call in the body of an invoked clause its effect is to cut out all the as yet untried clauses for the call which invokes the clause, and all the alternative untried evaluation paths for the calls that precede the / in the clause. Suppose that a clause of the form

```
((R ...)(R1 ...)(R2 ...)/(R3 ...))
```

is invoked by some R-atom call. The calls (R1 ...) and (R2 ...) are evaluated in the normal way with any necessary backtracking in order to find a solution to both calls. If they can both be solved, the / is executed. Slash always 'succeeds'; it is used for its side effect.

It suppresses all further backtracking on the evaluation of the (R1 ...) and (R2 ...) calls that would normally occur on a subsequent failure or when trying to find all the solutions to a condition. It also prevents any later as yet untried clauses for R from being used to solve the particular R-atom call that invoked the clause.

In other words, if the (R3 ...) call should now fail, then the call that invoked the clause also immediately fails. If the slash was not there, a failure of (R3 ...) would result in alternative ways of solving the (R1 ...) and (R2 ...) calls being explored, and then in alternative clauses for R being tried. The / cuts out all these alternatives.

The slash is useful for 'telling' micro-PROLOG that there are no alternative solutions to be found once the evaluation has reached the / point in the clause. This saves wasted time searching for non-existent alternative solutions, and also enables micro-PROLOG to save space.

The / actually side effects the evaluation stack. It discards all the activation records back to the activation record

for the invoked clause containing the /. This removes all the backtracking information for the evaluation of the calls that precede the / in the clause. Finally, it side-effects the backtracking information associated with the invoked clause so that it appears to be the last clause for the call. The / is also used in an essential way to implement some of the logical primitives of micro-PROLOG.

Used in a ? query a / prevents backtracking on the calls that precede it in the query once a solution to each of them has been found. This is because the ? is defined by

```
((? X)X)
```

and so a query

```
?(A1 .. Ak / ...)
```

becomes the evaluation of the body

```
A1 .. Ak / ...
```

for the single clause for ?. The / cuts out the backtracking options for A1 .. Ak. There are no other clauses for ? to be suppressed.

You can use a ? call when you want to cut out backtracking on preceding calls but not prevent the use of later clauses. Thus, the evaluation of the / in

```
((R ...) (? ((R1 ...)(R2 ...))) (R3 ...))
```

will cut out alternative solutions to (R1 ..)(R2 ..) but will allow the use of untried clauses for the invoking Rcall. A / placed inside an atom list argument to a logical primitive has a similar 'local' effect.

#### 7.12.4 FAIL

```
FAIL {false}
```

The FAIL predicate always evaluates to false. This is used to fail a branch of the proof, FAIL has no clauses, but the interpreter knows about it and does not signal a "No clauses for" error.

#### 7.12.5 ABORT

```
ABORT {abort the current supervisor command}
```

The ABORT primitive cancels the current supervisor command that is being executed and returns to the supervisor.

#### 7.12.6 /#

```
(/# t1 t2 .. tk) {t1 ... tk is true}
```

The /# is the comment predicate. It ignores its sequence of any number of argument terms (including none) and always succeeds with no side-effect. /# can be used within clauses to provide comments or as a 'no-op' relation.

Its definition is equivalent to the program

```
((/* X)X)
```

#### 7.12.7 SPACE

```
(SPACE x) {x is the no. of K bytes left in workspace}
```

The SPACE primitive returns in its single argument the number of Kilo-bytes(1024 bytes) of space that are currently left in the workspace.

Before actually returning this number it calls the internal garbage collector, which has the effect of making sure that all the known garbage is removed.

#### 7.12.8 <SUP>

```
<SUP> {the ever running supervisor program}
```

The supervisor control program can be listed as the program for the predicate symbol <SUP>. It is a very simple program with just a few clauses. Its main function is to add clauses that are entered and to call other programs invoked by commands.

```
(("<SUP>"
  (CMOD Y) {find the current module name}
  (P Y) {print module name as prompt}
  (R X) {read in next term}
  ("<SUP> X)/ {process term deterministically (the /)}
  {"<SUP>"}) {tail recurse - loop back to handle next
             command}

(("<SUP> X)
  (CON X) {read-in term is a constant i.e. a
           unary relation/command name}
  (R Y) {read in its single argument}
  (X Y)) {execute command by calling the relation}
```

```
((("<SUP> X|Y)) {read-in term is a micro-PROLOG clause}
  (ADDCL (X|Y))) {add it to the program}
```

```
((("<SUP> X)
  (PP ?)) {command fails, or illegal input}
          {display ?}
```

The <SUP> primitive is invoked as soon as the micro-PROLOG interpreter and the supervisor have been loaded. Its evaluation only terminates on a fatal error or a QI.

## 8. Adding assembler coded subroutines

It is part of the philosophy of micro-PROLOG that it should be as extensible as possible. This is reflected in the flexibility of the syntax, as well as in the inherent extensibility of PROLOG. A further kind of extension provided for in micro-PROLOG is the ability to add programs to the system that are written in other languages, in particular assembly coded programs, and have them automatically executed by the system like any other program.

To this end we have an interface which, if followed exactly, allows a 'foreign' program to be called by the system and parameters to be passed between it and micro-PROLOG. This interface is also used by the bulk of the built-in programs, so this chapter also gives a flavour of how they are implemented.

Any new built-in programs have to be written in Z80-assembler; while it is possible to add in a program written in FORTRAN (say) it would be necessary to sort out the machine code interface to allow the micro-PROLOG system to call the FORTRAN subroutine. This requires detailed knowledge of the particular FORTRAN compiler, and is beyond the scope of this manual.

A user coded program is invoked in the normal way, by an atom in a goal statement, or in a clause. Like the built-in machine coded programs the extent to which it behaves like a normal program, written as clauses, depends on how many of the program's uses have been catered for, though the interface only handles deterministic uses. If a non-deterministic use is to be handled, then it can be programmed up using a micro-PROLOG program that explicitly sequences through the non-deterministic choices.

The principal interface between micro-PROLOG and a user coded (or any other) machine language program consists of three components. A number of data registers are provided through which parameter values are passed between micro-PROLOG and the machine coded program. A type tree is used to specify what types of arguments the program can accept, how many of them, and what patterns of use are supported. The type tree also specifies the actual entry points into the program, so that depending on the particular call different entry points may be entered.

The third component of the interface is the predicate symbol declaration. This declares to micro-PROLOG a constant which describes the name to be used to access the program and its initial entry point. The predicate symbol has to have a different name from any already existing in the micro-PROLOG system.

### Case study

To illustrate the method for inserting a new program into micro-PROLOG we take as a case study a simple program which finds the length of a constant. We shall call the new built-in primitive LENGTHOF, and it is read as:

(LENGTHOF constant integer)



{the number of chars in constant is integer}

We shall allow two possible uses of the LENGTHOF program, the first is to count the number of characters in a symbol and return the result as an integer, and the second verifies that a given constant has the correct number of characters in its name.

### 8.1 Data registers

Each argument of a call to an assembler coded primitive is left in a **Data register** on entry to the call. There are eight of these registers provided in the system, corresponding to up to eight arguments in a call. No user coded program may have more than eight arguments, though the system does not check this. Of course most programs have considerably fewer than eight arguments, in which case not all of the registers are used. However, those that are not used must not be altered in any way by the user program.

Each data register is six bytes long, and has the format:

	B <sub>0</sub>	B <sub>1</sub>	B <sub>2</sub>	B <sub>3</sub>	B <sub>4</sub>	B <sub>5</sub>
Output						
value/						
pointer						
Input						
value/						
pointer						

The eight data registers are labelled SLOT1 .. SLOT8; their addresses can be found by examining the file PROLOG.SYM supplied on the micro-PROLOG distribution disk.

A Data register has two separate components, one for input data to the user program, and one for returning results. The output component is of the same format as a micro-PROLOG value cell.

The input component specifies what the argument to the built-in primitive is at the time of the call; and the output component is used to hold the returned values of the primitive. By assigning to the output component of the data register a value is returned to the PROLOG program which called the primitive. The output component is only significant if the input component holds a variable.

The input component of the data register is determined by the type tree (see below). It can be either a variable, number, constant or list pointer.

Generally, if the type tree has been followed the invoked program already knows what is contained in a given data register and can act accordingly without further checking. The user must under no circumstances affect the value in the input register: the value can be read, but not modified.

The output type field is initially set to OFFH. To pass back a value it is necessary to assign the type of the answer to this field, and to place the appropriate value in the value field. If the type field is left at OFFH then no value is passed back, and the corresponding variable is left unbound.

### 8.2 Value cells and terms in micro-PROLOG

A **value cell** consists of a one byte tag field, and a two byte integer or pointer value. Value cells are used internally by micro-PROLOG in, for example, list cells. Each value cell is effectively a term in the system; it can be a list, an integer, a floating point number, a constant or nil.

If it is a list then the value cell is a pointer to a pair of value cells (which must be on an even byte boundary); if it is an integer then the value cell contains the value of the integer as a 16bit two's complement number; if it is a floating point number then the value cell points to a pair of value cells in a special format (see below); if it is a constant then the value cell points to a constant data structure (see below); if it is nil then only the tag is important.

The various data types that are recognised by micro-PROLOG include:

NUMBER	=	4	Data field holds a 16 bit number in two's complement form.
FLOAT	=	10	Value is a pointer to a packed fpn
NIL	=	16	The empty list. Data field ignored.
CONSTANT	=	8	Data field points to a constant structure
LIST	=	3	Data field points to a list cell

#### 8.2.1 micro-PROLOG internal data structures

There are only three data structures used to construct terms in micro-PROLOG (apart from the scalars: integers and nil), these are list cells, constants and floating point numbers. We give a brief description of each of these. Note that there are actually other objects in the micro-PROLOG system, but these are not part of the general interface.

In fact it is suggested that built-in primitives do not directly manipulate list structures but only objects such as constants and numbers.

A Constant has a structure of the form:

Value Cell      Name of Constant FF

New constants can be made very simply, by filling a special buffer (CONBUF) within micro-PROLOG with the characters of the constant, terminated by a 0 byte, and calling the internal routine MAKCON. This routine returns in the HL register a pointer to the constant; if the constant already existed in the dictionary then the existing constant is returned; otherwise a new one is constructed.

A list cell consists simply of two value cells contiguous in memory, with the head cell first and on an even byte boundary.

Head Value Cell      Tail Value Cell

List cells are created by using the internal routine CONS which returns a pointer to a list pair in the HL register. The head and tail of the list are both initialised to nil.

Note that during the execution of CONS (and MAKCON) the

## 8. Adding assembler coded subroutines

garbage collector may be called. It is imperative that the system be made aware of any lists you are constructing. This is quite simple to do: construct the list from the outside in (i.e. construct the outermost list pair first, then build the inner parts all the while linking them to the outer list pair); and assign the output part of one of the data registers to the top list pair constructed. This data register does not have to be one of the ones used to pass arguments; though if you do use another one be sure to reset the output type field back to 0ffh before returning to micro-PROLOG.

Floating point numbers come in two flavours, packed and unpacked. A floating point number has to be unpacked before any computations may be performed on it, and re-packed if it forms part of the output value of the primitive.

A packed floating point number occupies the same space as a list cell, and are garbage collected along with other list cells. In the 6 bytes available are packed the fraction field (8 digits packed BCD), the exponent field (one byte containing exponent as 10-127..10127 using offset 128 (80H)).

The sign of the fraction is held in bit 3 of the first byte of the cell, bit3=1 means negative, bit3=0 means a positive number.

The fraction is held as 8 consecutive digits, i.e. four bytes of packed BCD. The most significant digit is the upper nibble of the first byte, and the least significant digit is the lower nibble of the last byte. The format of the floating point number looks like:

R

```
Sign Exp d0d1 d2d3 d4d5 d6d7
B0 B1 B2 B3 B4 B5
```

The floating point number thus represented is

```
Sign * 0.d0d1d2d3d4d5d6d7 * 10Exp-128
```

For computation on floating point numbers (fnp's) a special unpacked layout is used. This is similar to the packed form, except that each digit of the fraction part occupies a single byte:

```
Sign Exp d0 d1 d2 d3 d4 d5 d6 d7
B0 B1 B2 B3 B4 B5 B6 B7 B8 B9
```

Also it is wise to leave some free bytes (9 bytes) at the end of an unpacked number to allow for unnormalised numbers and excess numbers from results of multiply & divide.

To pack a floating point number, call CONS to create some space, and then call PACK to load the value into the space created.

### 8.2.2 Warning

There are other types recognised by micro-PROLOG, but all other values are reserved by micro-PROLOG. The type field is checked at various points in the system, and an invalid type may result in micro-PROLOG being aborted.

In fact it is envisaged that the type most commonly used by

## 8. Adding assembler coded subroutines

user programs is that of number. It is for the sake of completeness that the other types have been described.

### 8.3 Type tree

Type checking of arguments to a machine coded program is controlled by the type tree. This is a data structure that is part of the program, and must be provided with it. Only if no arguments are expected to a call may this tree be omitted, but if the programmer wants the system to check that it is called with no arguments then a tree can be specified to check for this.

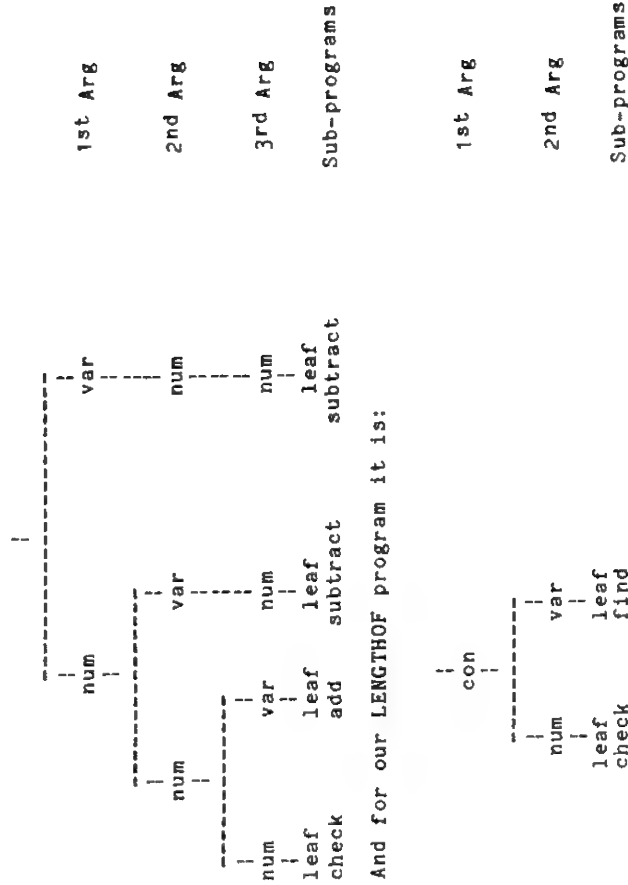
Each node in the tree has 5 fields, corresponding to the possible types that an argument in the call can have. Each different type of argument leads to a subtree of the type tree, with an empty subtree signifying that a particular type of argument is not allowed. The empty subtree is marked by having the value 0FFH in the corresponding field of the node. The depth of the tree corresponds with the argument position in the call: the root of the tree deals with the first argument, and the nodes in the second level (i.e. those immediately descended from the root) deal with the second argument position.

Type tree node:

Leaf Num Con List Var

B0 B1 B2 B3 B4

For example the type tree for the SUM predicate may be represented as follows:



The **Leaf** field refers to the end of the argument list: i.e. no argument. The subtree rooted at the leaf field is actually an entry point into the code of the user program proper. At this point all of the parameters to the call will have been parsed and the appropriate values placed in the data registers. Furthermore, since the path from the root node of the tree to the entry point is unique the code can simply access the values in the knowledge that the types are as expected. All that is left for the program to do is to compute the answer values, place the result in the output halves of the data registers and return.

When returning from a leaf program (by executing a RET instruction) the system checks the return code for success or failure. If the Z flag is set then the system assumes success, and the variables are bound as specified, if however, the Z flag is reset then the call is assumed to have failed. In this case the micro-PROLOG system backtracks in the normal way.

The **Num** field has rooted from it a non-empty subtree if a number was allowed in the current argument position. If a number is present, and the number subtree is non-empty then the number in the call is loaded into the appropriate data register, the number subtree followed and the next argument considered.

Note that to decide whether an integer or floating point number is present the input type byte will have to be examined in the relevant data register. If it is an integer the input type value will be 4, if a floating point number it will be 10.

If the **Con** subtree is non-empty then a constant is allowed as a legal argument. If a constant appears as an actual parameter then the constant's address is loaded into the input data register.

If the **List** subtree is non-empty then a list is allowed as an actual parameter. Note that this means that a non-empty list as well as the empty list is allowed. If a list is encountered as an actual parameter then a pointer to a value cell which points to the list (or has NIL as a type) is placed in the data register: **NOT** a pointer to the list itself.

If the **Var** subtree is non-empty then a variable is allowed as an actual parameter. If a variable is used where one is not allowed then a "CONTROL ERROR" is reported, similarly if only a variable is allowed but a variable not used as an actual parameter, then a "CONTROL ERROR" also results. The variable subtree is used when the programmer expects to return a result in that argument position, although there is no actual compulsion to return a value. Note that, of course, values cannot be returned other than through a variable!

Each field in the node is a single byte unsigned number in the range 5..255. If a non-empty subtree is rooted at a particular field then the number in the field is a relative offset: it is the distance, in bytes, between the target node or entry point and the base of the current node. If an actual parameter is of a type which has no legal subtree for it, for example if a number is supplied as a parameter but a number is not allowed, then the call fails, unless the actual parameter was a variable, or a variable was the only type allowed.

To set up the type tree interface, for our LENGTHOF program for example, we must start the initial entry point of the program as follows:

```

LENGTH:  ORG <BOS>           ;Where <BOS> is the contents of BOS
          LD IX,LENTRE       ;load IX with type tree for this prog
          JP TRWALK          ;entry point inside micro-PROLOG
                               ;which processes the type tree

```

The type tree for the LENGTHOF program is:

```

LENTRE:  DEFB 255,255,LTR2-LENTRE,255,255
LTR2:    DEFB 255,LTR3-LTR2,255,255,LTR4-LTR2
LTR3:    DEFB LCHECK-LTR3,255,255,255,255
LTR4:    DEFB LFINL-LTR4,255,255,255,255

```

The main program for LENGTHOF is:

```

INDATA EQU 3           ;define the relevant offsets in
INTYPE EQU 5           ;the data registers
OUTDTA EQU 1           ;output value cell
OUTYPE EQU 0

;NUMBER EQU 4           ;type value for integer

;LEN:  LD HL,(SLOT1+INDATA) ;look at the input constant
       INC HL              ;move to the text part
       INC HL
       INC HL
       LD BC,0
       LD A,Offh
       CP (HL)
       RET Z
       INC HL
       INC BC
       JR LOOP

;LCHECK: LD A,(SLOT2+INTYPE) ;check the length
         CP NUMBER          ;is the given an integer?
         RET NZ             ;can't be non-integer length
         CALL LEN           ;find out the actual length
         LD HL,(SLOT2+INDATA) ;see what it's supposed to be
         OR A
         SBC HL,BC
         RET

;LFIND:  CALL LEN          ;find out how long it is
         LD (SLOT2+OUTDTA),BC ;store the result in register
         LD A,NUMBER       ;it's an integer
         LD (SLOT2+OUTYPE),A
         CP A
         RET
         ;set Z for success
         ;all constants have a length

```

#### 8.4 Predicate symbol declaration

The predicate symbol declaration is used to describe the name of the new program, and its initial entry point, to the micro-PROLOG system. The predicate symbol is defined by a constant structure like that seen above. Note that the

declaration of a constant is not in itself sufficient, since micro-PROLOG does not yet 'know' about it. The new constant has to be patched into the system dictionary before the program can be used.

The constant declaration for our RND program can be coded in assembler as follows:

```

LANGTHF: DEFB 4           ;Number type to signify M/C code
          DEFW LENGTH      ;Initial entry to LENGTHOF program
          DEFM 'LENGTHOF'   ;Text of name of predicate symbol
          DEFB OFFH        ;Byte terminator of name string

```

The entry in the dictionary takes the form of a list cell, and can be coded as:

```

NEWDCT: DEFB 8           ;Constant type
          DEFW LANGTHF    ;Point to new constant
          DEFB 3          ;List type
          DEFW <SDICT>     ;Point to top of system dictionary

```

Where <SDICT> is the contents of SDICT

This completes the definition of the LENGTHOF program; all that is needed now is to assemble it for the right place, and link it in to micro-PROLOG. Until the new predicate symbol is inserted into the system dictionary it cannot be referenced by a micro-PROLOG program.

### 8.5 Inserting a program

In the micro-PROLOG system there are two pointers which are necessary to adjust and know about when inserting a new program. The first is BOS, which points to the first available byte of memory. The second is a pointer to the top of the system dictionary SDICT.

When micro-PROLOG is loaded into memory the Transient Program Area (TPA) of CP/M looks something like:

```

micro-PROLOG      Available RAM
100H              BOS

```

The BOS pointer points to the first available byte in memory above the main micro-PROLOG system. micro-PROLOG builds its data structures above this pointer. The new LENGTHOF primitive has to be added in to the micro-PROLOG system at or above the BOS pointer. This pointer has also to be adjusted to point above the end of the new primitive.

The procedure for adding a new program to micro-PROLOG (once it is assembled into the right location) involves loading the new program starting at (BOS), updating the BOS pointer, and updating the system dictionary pointer.

### 8.6 Extending micro-PROLOG

It will be appreciated that any one attempting to augment micro-PROLOG by adding new assembler built-in programs should be reasonably proficient in (a) programming in assembler, (b) interfacing to CP/M and (c) using micro-PROLOG.

Finally, be very careful as it is very easy to damage micro-PROLOG. (Never modify in any way the original distribution disk.) The interface described above is a very simple and powerful one; it is used by the great majority of the built-in programs in standard micro-PROLOG.

Adding built-in programs does not invalidate the licence agreement; however it is not permitted to sell or otherwise distribute an augmented version of micro-PROLOG without the written permission of the copyright holders of micro-PROLOG. Of course any augmentations that you build are not the property of the copyright holders.

## Appendix A

### Error messages and error handling

micro-PROLOG has relatively few error conditions, and most of them can be trapped by a micro-PROLOG program. The errors are divided into two groups - the numbered errors and the message errors. All the numbered errors can be trapped. When a numbered error occurs, if there are clauses for the relation "ERROR?", the program for the relation is called with a call of the form:

```
("ERROR?" <error number> <error call>)
```

This call replaces the error call passed as the second argument, this is the call that was being evaluated when the error occurred. Thus, if the call to the "ERROR?" program succeeds, possible binding variables in its error call second argument, the error call is assumed to have succeeded and the evaluation continues with the next call. If the call to the "ERROR?" program fails, the error call is assumed to have failed. The call to the error handler may also result in an ABORT to the supervisor.

If there are no clauses for the "ERROR?" relation, the message

Error: n

where n is the error number, is displayed. The current evaluation is then aborted and you are returned to the supervisor.

#### The numbered errors

The errors, and their numbers, are:

- |   |  |
|---|--|
| 0 | <b>Arithmetic overflow.</b> If a call to an arithmetic primitive results in a number which cannot be represented then an arithmetic overflow is signalled. This includes the case of division by zero.   |
| 1 | <b>Arithmetic underflow.</b> This error arises when a number becomes too small to represent (the exponent becomes too negative).   |
| 2 | <b>Clause error.</b> There are no clauses defined for the call being executed. Some PROLOG systems merely fail the call if there are no clauses for its relation. In micro-PROLOG you can simulate that behaviour by having a special clause in the "ERROR?" program to FAIL the "ERROR?" call for error number 2. |
| 3 | <b>Control error.</b> The built-in primitives in micro-PROLOG often require a minimum number of arguments  |

## A. Error messages and error handling

to be given at the time of the call. If the arguments to a call to such a primitive are underspecified, this error occurs. The error is also signalled if the evaluation has reached the use of a meta-variable (see Chapter 2) and the meta-variable is unbound or has a value of the wrong form. There is a strong possibility that the calls in some clause or query are incorrectly ordered, and that the call that would bind the unbound variable has not yet been evaluated.

4 **ADDCL error.** This error is signalled when you try to add a clause for a relation name which is either a primitive of micro-PROLOG, or is imported to the current module, or is the name of a currently opened file.

5 **File error.** This error is signalled when an error arises during a file operation. For example, if you try to open a file using a file name which is also the name of a program relation you will get this error. Other examples include CREATING a file already open, trying to SEEK to one of the special serial files like "CON:", or trying to SEEK to an unopened file.

6 **Too many files error.** micro-PROLOG only allows upto four files to be opened at any one time. If you try to open more than four files you will get this error message.

7 **Opened files error.** You will get this error if you try to LOGIN a new disk with files on the current disk still open. The LOGIN is supposed to allow disks to be changed; so all files on the disk to be replaced have to be closed before the LOGIN.

8 **Bad data error.** You will get this message if you position into a file using SEEK to a position in which there is no data and you then try to read from that position. Other instances of bad data on the disk are trapped by CP/M, and CP/M does not allow recovery. It will abort micro-PROLOG.

9 **Write error.** You may get this error when a write error is detected by CP/M. In fact, you probably will never get the error since CP/M nearly always aborts micro-PROLOG in such circumstances.

10 **Disk full.** You get this error if the disk becomes full, or no more directory space is available for a new file.

11 **Break!** This error is signalled when the user types ^C at the console.

12 **Module error.** This error is signalled whenever there is an illegal use of modules. It occurs when you try to CRMOD or LOAD a module other than at the root module level, or if the new module has

## A. Error messages and error handling

a relation name in its export list and the relation is already defined by some program.

The following program is an example of a simple error handler which just reports the error with a short message and then aborts to the top-level supervisor:

```
(("?ERROR?" X Y)
(P-code X Z)/
(P Z Y)(PP)
ABORT)
((P-code 0 "Arithmetic overflow"))
((P-code 1 "Arithmetic underflow"))
((P-code 2 "No clauses for"))
((P-code 3 "Control error"))
((P-code 4 "Error in adding clause"))
((P-code 5 "File error"))
((P-code 6 "Too many files opened at once"))
((P-code 7 "Close all files first"))
((P-code 8 "Bad data in file"))
((P-code 9 "Write error"))
((P-code 10 "Disk or directory is full"))
((P-code 11 "Break !"))
((P-code 12 "Illegal use of modules"))
((P-code X (Error X)))
```

A simple alteration will allow the error 2 to be treated as a failure. All that is needed is to add the following clause to the ?ERROR? program:

```
(("?ERROR?" 2 X)
/
FAIL)
```

This must be placed before the main ?ERROR? clause. The the / followed by the FAIL causes the error call given as the X argument to fail.

### Message errors

There are some errors from which it is not possible to recover. When these errors occur, a message is displayed and there is a default effect that cannot be changed. The error messages and the effects are:

**Dict error**

There is no space left in the dictionary for new constants. The evaluation is ABORTed and you are returned to the supervisor.

**Out of space**

Garbage collection is not able to free enough space for the current evaluation to continue. You are ABORTed to the supervisor.

**File not found**

There has been an attempt to OPEN a file not on the currently logged in disk. The OPEN call is failed but the evaluation continues.

**Syntax error**

Occurs when a term is being read in. It is displayed if there are too many right brackets or there is more than one term following a !. In

## A. Error messages and error handling

either case the read continues with the extra brackets or terms ignored.

**System Abort** This arises when micro-PROLOG detects an internal inconsistency within the system. It means that a vital internal data structure has been destroyed and micro-PROLOG cannot precede. You are aborted out of micro-PROLOG to CP/M.

You should actually never get a system abort, though there are situations where programmer action can cause one. The simplest case where a programmer can cause a system abort is in trying to execute the following program:

```
((abort-micro (x))
 (abort-micro x))
```

(This causes overflow in the garbage collector and cannot reliably be recovered from)

Another situation which can cause system aborts is in certain uses of DELCL. Unlike many PROLOG systems micro-PROLOG does actually delete a clause from the main internal dictionary structures when a DELCL primitive is invoked.

However a clause can be deleted while there are references to it if the clause is in use during the current execution.

If you only delete clauses while at the top level through one of the front end programs such as SIMPLE and MICRO then you are always safe. However, if you delete a clause in a dynamic way then you may get a problem, in general situations like:

```
(Pred .. ) ... (DELCL ((Pred ...))) ... FAIL
```

are likely to be dangerous. In this situation you may or may not get a system abort depending on whether the garbage collector is called.

## B. Useful addresses

The file PROLOG.SYM which is supplied on the distribution disk gives the actual numerical values of a number of internal addresses within the micro-PROLOG system. These addresses are of use when adding a new built-in primitive to micro-PROLOG, or when altering the lexical type tables (see Appendix C).

In this appendix we list the labels which are relevant and give a brief description of each.

SOLVE	Warm start entry point to micro-PROLOG
ERREXT	Error entry point HL contains the error code
TRWALK	The type tree walker
SLOT1	The first data register
SLOT2-8	The second to eight data registers
\$MEMORY	Word pointer to end of micro-PROLOG
BOS	Same
UDICT	User dictionary
SDICT	Pointer to top of system dictionary (two bytes)
PACK	pack a fpn      HL = address of packed real number DE = address of unpacked fpn
UNPACK	unpack a real pointer      ditto
FADD	Add two unpacked numbers      BC:=HL+DE
FDIV	Floating point divide      BC:=DE/HL
FINT	Find nearest integer      DE:=floor(HL)
FLOAT	Convert integer to fpn      DE:=float(HL) HL = integer
FMINUS	Negate floating point number      HL:= -HL (unpacked)
FMULT	Multiply two fpn's together      BC:=HL*DE
FNORM	Normalise a fpn      HL = address of fpn
FTEST	Compare two fpn's      Z:= HL=DE
FZERO	Test to see if fpn is zero      Z:= HL=0
COLLECT	Garbage collector (preserves all registers)
MAKCON	Create a new constant. CONBUF contains the text, HL is left with a pointer to the new constant
CONBUF	A 64 byte area used to construct new constants (text terminated by a null byte)
CONS	Leaves in HL a pointer to a new list cell
CHROUT	Send A register as a character to console
CRLF	Throw a new line on the console
INTCHK	Poll the console for interrupts
INKEY	Reinitialise the console buffer
GETCH	Universal get character jump for term input
BAKCH	Universal character backup for term input (1 char only)
SENDCH	Universal send character jump
RSETUP	Use first argument of predicate to set up input stream
WSETUP	Use first argument to set up write stream



## B. Useful addresses

PRTERM Universal term output (through SENDCH)  
 RDTERM Universal term input (from GETCH & BAKCH)  
 WNUM Signed 16bit number print (through SENDCH)  
 WREAL Display floating point number (through SENDCH)

LEXTAB Lexical definition table  
 NOVARS No. of variable prefix characters  
 ERRCHR Character used for out of range variable names

## Appendix C

### Changing the lexical rules

The micro-PROLOG tokeniser is a program that separates the sequence of characters in the input into tokens. It is driven by a table, which is called LEXTAB, which describes the character set in terms of different subsets: the separator characters, the special characters, the digits, the letters, the graphic characters, the sign and quote characters, and the variable prefix characters.

Each character's membership of these subsets is represented by a single byte in the table, with each bit in the byte representing a different set. If the appropriate bit in the byte is on (i.e. 1), then it signifies that the character belongs to that set.

By modifying these subsets the lexical rules can be made to look very different; for example by merging the graphical and letter sets into one (the letter set) then the distinction that micro-PROLOG makes between the two sets is ignored. This would allow such tokens as:

```
$A X1 A'B
```

However, the subset most likely to be of interest is the variable prefix character set. This subset defines the conventions that micro-PROLOG uses to distinguish variables from constants. In standard micro-PROLOG the variable prefix subset is:

```
{x y z X Y Z}
```

Tokens beginning with these letters may be recognised as variables. By changing this set we can implement different conventions for variables. This approach is a response to the current multiplicity of ways of recognising variables.

To implement a convention where identifiers whose first letter is a lower case letter followed by digit characters are recognised to be variables (similar to IC-PROLOG [Clark & McCabe 1979]) all that is necessary is that the variable prefix character set be changed to:

```
{a b .. z}
```

To implement the DEC-10 convention of upper case variables, lower case constants the variable prefix set should be changed to:

```
{A B .. Z}
```

Finally to implement the convention, found in the original Marcellus PROLOG and in Waterloo PROLOG, of using the character # in front of a token to signal a variable, the variable prefix character set should be:

```
{#}
```

## C. Changing the lexical rules

Note that in this case the character # will also have to be made a letter.

Apart from reading variables, it is necessary to print them, preferably in the format that variables are read in. In micro-PROLOG all variables are printed with a variable prefix character (possibly) followed by a sequence of digits.

This is of course the kind of token that would subsequently be read as a variable. The prefix character used is taken from the table of lexical types, each variable prefix character defined in the table will be used when printing variables, in the order that they appear in the table. Thus the first seven variables (in standard micro-PROLOG) are printed as:

```
X Y Z x y z X1 ...
```

To actually change types of the various characters it is necessary to use the CP/M utility ddt to modify the PROLOG program. This utility is a general debugging package and is part of standard CP/M. We need to examine and modify certain locations in PROLOG. ddt is executed by using the CP/M command:

```
A> ddt prolog.com
```

This loads the PROLOG system into memory and enters the command mode of ddt. The initial response of ddt is like:

```
DDT VER 2.2
NEXT PC
3f80 0100
```

Be careful to note down the number under NEXT as it is needed later. ddt works entirely in hex arithmetic. The various addresses we use below are absolute memory addresses, as this is how CP/M is organized; in particular you should be careful about modifying memory locations other than described below as ddt allows you to change any memory location including ddt and CP/M.

The table LEXTAB consists of 128 single byte entries. Each character in the ASCII character set has an entry associated with it; the entry is found by adding the value of the character to the base address of LEXTAB.

Each bit in the entry corresponds to one of the subsets discussed above, if the bit is on then the character is said to belong to the appropriate set, if off then the character does not belong to the subset in question. Note that the two characters corresponding to 0 (Null) and 127 (Del) are illegal, and belong to none of the subsets.

Each byte in LEXTAB is organised as:

```
Graphic character
Digit character
Letter
```

```
7 6 5 4 3 2 1 0
```

Variable prefix character

The other bits in the byte have significance, but it is not necessary to know of them; be careful not to disturb these bits

## C. Changing the lexical rules

as unexpected results may occur.

Some example entries of LEXTAB are:

Char	Hex	SYSQGSDDL	Hex
A	41H	00000001	= 01H Letter
x	6DH	01000001	= 41H Var. prefix & Letter
0	30H	00000010	= 02H Digit

So, to implement our # convention for variables we have to change the table entries for #, X, Y, Z, x, y & z. The six letters have existing table entries of 41 (hex) which reflects that they are both letters and variable prefix characters; these entries have to be changed to be just letters. The letters are in two groups of three successive bytes in the table: X, Y & Z and x, y & z.

The table entry for any character can be found by adding the value of the ASCII representation of the character to the base address of LEXTAB (whose address can be found by examining the file PROLOG.SYM supplied on the system disk). For example X in the 88th character in the ASCII sequence, (which is 58 in hex), so the entry for X is <LEXTAB> + 58 (hex). We can use the ddt h command to do this hexadecimal arithmetic for us, suppose that LEXTAB were located at 0124H:

```
-h58.124
017C FF34
```

The first number printed is the address in memory (ignore the second number) of the table entry for X, the entries for Y and Z immediately follow it. To modify the entry we use the ddt s command. This command enables memory locations to be modified in sequence. The old value of the byte is printed, and it is changed by entering (in hex) the new value of the byte. After carriage return is pressed the next memory location is examined, allowing it to be changed too. The s command is terminated by using . instead of the new value of a byte. To change the X, Y and Z entries we can type as in:

```
-s17C
017C 41 1
017D 41 1
017E 41 1
017F XX 1
```

The x character is 78 (hex) in the ASCII code, so to change x & y and z to be just letters we do:

```
-h78.124
019C FF54
-s19C
019C 41 1
019D 41 1
019E 41 1
019F XX 1
```

Now we have to declare the # character as a variable prefix character. We must also change it from being a graphic character to being a letter, this is so that the tokeniser treats #123 as a single alpha-numeric token. The # character must

## C. Changing the lexical rules

therefore have the code 41 (hex) as its entry in LEXTAB. To change the entry we do what we did for the letters, we add the ASCII value of \* to the base address of LEXTAB and use the s command to change the entry byte. Now \* has ASCII value 2A (hex), so the required entry is computed by:

```
-b124.2A
014E 00FA
```

And we change the entry by:

```
-s14E
014E 08 41
014F XX +
```

A special counter (NOVARS) contains the number of variable prefix characters in the table that are to be used when printing variables. This single byte counter should be changed, if the variable prefix character set is changed, to reflect the number of prefix characters. This number should **never** be greater than the actual number of variable prefix characters in the table, and it should also be at least one. So we have to change this location to 1 (one) which is the new number of variable prefix characters:

```
-s124
0124 06 1
0125 00 +
```

This completes the changes to make the variable convention followed by digits. All that is now required is to exit ddt and save the memory image in the file PROLOG.COM, which has the effect of updating the old PROLOG system with the changes. ddt is exited by typing:

```
~C
A> Save 63 prolog.com
```

The number in the save command is found by converting the original NEXT value printed out by ddt. When ddt is first entered it gives the length of PROLOG in pages. The save command expects this number in decimal form, whereas ddt displays it in hex, so you have to convert it.

### Warning

Some of the entries in LEXTAB should not be changed. In particular the characters that are special should not be removed from the special set (otherwise the syntax analyser may not be able to recognise terms properly), and no new characters should be added to the special set.

Furthermore, it is only safe to modify the 'normal' character sets: letter, graphic and variable prefix characters. Any other alteration is likely to lead to problems.

Secondly, you should be careful about choosing which letters you use as variable prefix characters, since the READ program converts single letter tokens into variables if the single letter is a variable prefix character. This means that certain programs which expect single letter responses (such as the editor which uses e, u and w (among others)) to be constants may have to be modified. In particular if the letters u, v and w were added

## C. Changing the lexical rules

to the standard set of variable prefix characters then the editor will have to be changed to use something other than u and w for the unwrap and wrap commands.

Note that the various PROLOG programs supplied with micro-PROLOG will also have to be changed to reflect the new variable conventions you have implemented. This has to be done using a conventional text editor, such as the CP/M ed editor.

## Appendix D

### System requirements

The following configuration of equipment is required to run micro-PROLOG:

Z80 micro-computer (speed not critical)

CP/M 2.0 or later (will not run in CP/M 1.4)

At least 40K, preferably 64K memory

One floppy disk drive (two is handier)

keyboard with full lower case character capability, must be able to generate the | (vertical bar) key

console screen with non-destructive backspace

auto line feed on lines >80 characters in length (or whatever the line length of the console is)

These requirements are subject to alteration as new versions of micro-PROLOG for different computers are introduced.

## References

- Clark, K.L., [1978], Negation as Failure. Logic and Data Bases, (H.Gallaire and J.Minker, Eds.), Plenum Press, New York, pp. 293-322.
- Clark, K.L., McCabe, F., [1979], Control facilities of IC-PROLOG. Expert systems in the Micro-Electronic Age. Ed D.Michie Edinburgh Univ.Press.
- Clark K.L., Ennals, J.R., McCabe, F., [1981], A micro-PROLOG Primer, Logic Programming Associates Ltd.
- Clocksia W.F., Mellish C.S., [1981], Programming in Prolog, Springer-Verlag, New York.
- Colmerauer, A., [1973], Les systemes-Q ou un Formalisme pour Analyser et Synthetiser des Phrases sur Ordinateur. Publication Interne No.43, Dept. d'Informatique, Universite de Montreal.
- Colmerauer, A., [1978], Metamorphosis Grammars. Natural Language Communication with Computers, (L. Rolc, Ed.), Lecture Notes in Computer Science No. 63, Springer-Verlag, pp. 133-189.
- Kanoui H., Van Canaghem M., [1980], Implementing a very high level language on a very low cost computer. Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille, Luminy.
- Knuth D.E., [1968] The Art of Computer programming. pp 147-151. Addison Wesley. Volume II, Semi-numerical algorithms.
- Kowalski, R.A., [1974], Predicate Logic as Programming Language. Proc. IFIP 74, North Holland Publishing Co., Amsterdam. pp. 569-574.
- Kowalski, R.A., [1979], Logic for Problem Solving. Artificial Intelligence series, North Holland Inc., New York.
- McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., Levin, M.I., [1962], LISP Programmers Manual. MIT Press. Cambridge, Mass.
- Moss, C.D.S., [1979], A New Grammar for Algol 68. Dep. Rep. 79/6, Imperial College, London.
- Naur, P., ed. [1962] Revised Report on the Algorithmic Language Algol 60. IFIP 1962
- Roberts G.W., [1977], An implementation of PROLOG. MSc thesis. Waterloo, Ontario, Canada.
- Robinson, J.A., [1965], A Machine Oriented Logic Based on the Resolution Principle. J. ACM 12 (January 1965), pp. 23-41.

## References

- Robinson, J.A., [1979], Logic: Form and Function. Edinburgh Univ.Press.
- Roussel, P., PROLOG: manuel de Reference et d'utilisation Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille, Luminy, Sept. 1975.
- Szeredi, P et al. MPROLOG user's manual. Institute for coordination of computer techniques. Hungary. 1368 Budapest, P.O.B 224.
- Van Emden, M.H., Kowalski, R.A., [1976], The Semantics of Predicate Logic as a Programming Language. J. ACM, Vol 23, No 4, pp. 733-742.
- Pereira L, Pereira F & Warren D. [1978], User's guide to DEC system 10 PROLOG. Dept AI, University of Edinburgh.

ch.  
painted